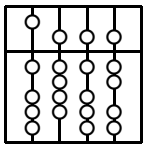
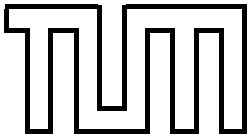


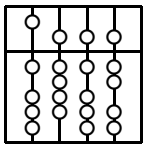
Institut für Informatik der
Technischen Universität München



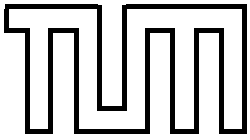
Experience-Based Control and Coordination of Autonomous Mobile Systems in Dynamic Environments

Dissertation

Sebastian Buck



Institut für Informatik der
Technischen Universität München



Experience-Based Control and Coordination of Autonomous Mobile Systems in Dynamic Environments

Sebastian Buck

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Alois Knoll

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Bernd Radig
2. Univ.-Prof. Dr. Günther Palm,
   Universität Ulm

Die Dissertation wurde am 25.03.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.12.2003 angenommen.

Abstract

Many real-time machine control skills are too complex and laborious to be coded by hand. Preferably, such skills are acquired by learning algorithms. Suitable algorithms should learn automatically and based on experience from interaction with the machine's environment. But unfortunately, typical learning methods for real world machine control tasks have a number of problems: Huge high-dimensional state spaces complicate inductive learning, and it might be difficult to get a sufficient amount of appropriate training data for learning either because it takes too long or because it is extremely difficult to obtain good examples for learning from exploration. Furthermore, most current learning algorithms rely on a discrete MDP-model of the continuous state space, suffer from the incremental summation of errors during learning, and neglect the existence of undesirable states.

The idea behind our approach of experience-based control is to exploit trajectories of successful explorations to approximate a value-function for the state space. To overcome the lack of training data we employ a realistic neural simulation of the machine's dynamics and introduce adequate exploration techniques, such as backward exploration, to acquire learning data. The combination of different exploration techniques allows for the integration of various types of initial knowledge and undesirable states can be integrated in the learning model. Since the majority of machine control tasks in technical applications shows deterministic behavior – or at least a unimodal probability distribution with a small variance – it is possible to use a simple projection-function instead of a complex MDP-model that was originally designed for discrete states. Our algorithms operate directly in a continuous state space and perform a number of explorations before we exploit the data. This is the main reason why our approach is robust against the incremental summation of noise which is often encountered in conventional learning algorithms. For the practical and efficient approximation of continuous functions we employ neural networks and networks of radial basis functions. Our methods have successfully been applied to numerous navigation tasks and tasks of situation dependent algorithm-selection.

Zusammenfassung

Viele Maschinensteuerungsaufgaben sind so komplex, dass es zu aufwändig wäre, sie von Hand zu programmieren. Im Idealfall wird hier das gewünschte Verhalten durch Lernalgorithmen erreicht. Geeignete Algorithmen müssen automatisch und basierend auf Erfahrungen aus der Interaktion mit der Umwelt der Maschine lernen. Leider zeigen viele gängige Lernalgorithmen für reale Maschinensteuerungsaufgaben einige Probleme: Sehr große und hochdimensionale Zustandsräume erschweren induktives Lernen, und es kann schwierig sein, eine ausreichende Menge geeigneter Trainingsdaten zu bekommen. Ursache dafür kann einerseits ein Mangel an Zeit sein; andererseits ist es vielleicht schwierig, überhaupt gute Beispiele zum Lernen zu finden. Darüber hinaus basieren die meisten gebräuchlichen Lernalgorithmen auf einem diskreten MDP-Modell des kontinuierlichen Zustandsraumes, leiden unter der inkrementellen Summierung von Fehlern während des Lernens und vernachlässigen die Existenz von unerwünschten Zuständen.

Die Idee, die dem vorgestellten Ansatz für erfahrungsbasierte Regelung zugrunde liegt, basiert auf der Ausnutzung von Trajektorien erfolgreicher Explorationen zur Approximation einer Bewertungsfunktion für den Zustandsraum. Um auch mit wenigen Trainingsdaten zum Erfolg zu gelangen, wird eine realistische neuronale Simulation der Dynamik der Maschine verwendet. Weiter werden intelligente Explorationstechniken wie z.B. Rückwärtsexploration eingesetzt, um an Trainingsdaten zu gelangen. Die Kombination verschiedener Explorationstechniken erlaubt die Integration verschiedensten initialen Wissens, und unerwünschte Zustände können vorab spezifiziert werden. Da die Mehrheit der technischen Maschinensteuerungsaufgaben deterministisches Verhalten – oder zumindest eine unimodale Verteilung mit kleiner Varianz – zeigt, ist es möglich, das komplexe MDP-Modell, das ohnehin für diskrete Zustände entwickelt wurde, durch eine einfache Projektionsfunktion zu ersetzen. Die vorgestellten Algorithmen arbeiten direkt in einem kontinuierlichen Zustandsraum und führen eine Anzahl von Explorationen durch, bevor die gesammelten Daten zum Lernen eingesetzt werden. Das ist auch der Hauptgrund, warum der vorgestellte Ansatz gegen die inkrementelle Summierung von Fehlern robust ist, die in konventionellen Lernalgorithmen weit verbreitet ist. Zur praktikablen und effizienten Approximation kontinuierlicher Funktionen werden neuronale Netze und Netze von radialen Basisfunktionen eingesetzt.

Die vorgestellten Methoden wurden erfolgreich in mehreren Navigationsaufgaben sowie in der situationsabhängigen Algorithmenauswahl eingesetzt.

Acknowledgements

First, I would like to thank my advisor, professor Bernd Radig, for his guidance during the last years and providing all the necessary requirements and support to write this dissertation. I am grateful to professor Günther Palm for becoming the second supervisor of this dissertation and for providing valuable support.

Very special thanks go to Michael Beetz for many prolific discussions. Without his ideas and our discussions, this work would not have come to fruition.

I wish to thank my former supervisors at University of Karlsruhe, professor Martin Riedmiller and Rainer Malaka for introducing me to machine learning and nurturing me through my graduate career.

Thanks go also to Gerhard Kraetzschmar, the Ulm Sparrows RoboCup Team, and the CS Freiburg RoboCup Team for motivating and funny RoboCup-related discussions and interesting training games.

Further thanks to Daniel Kiecza, Freek Stulp, and Andreas Hofhauser for reading drafts of this dissertation and providing valuable criticism. Their tireless input definitely improved the scientific and linguistic quality of this dissertation. Thanks to Robert Hanek for being a most enjoyable room mate and partner in scientific and non-scientific discussions. Furthermore I thank all people, researchers and students, involved in the AGILO-RoboCuppers, and especially Thorsten Schmitt, for an ongoing pleasant and successful collaboration. Thanks also go to Matthias Hilbig for helping me to solve the bureaucratic and formal tasks of this dissertation.
Altogether I want to thank my colleagues and former colleagues at Informatics IX for providing a supportive and most enjoyable working environment.

Finally, I wish to thank my parents, my wife Dörte, and my sons Ilian and Linus for all their support. This made my dissertation possible.

This work was supported in part by *Deutsche Forschungsgemeinschaft* (DFG, Semiautomatischer Erwerb visuomotorischer kooperativer Pläne) and in part by the *Siemens Mobile* Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of intelligent systems that can solve complex control tasks, both in industrial applications and in our daily life, is expected to be one of the key achievements of science and engineering in the first half of the current century. Applications such as autonomous cleaning robots, unmanned operating trains and airplanes, cars that can perform their driving operations in a self-directed manner, as well as the sophistication of recent space missions mark just the beginning of a new age of automation. All of these tasks demand reliable and accurate control algorithms that can cope with the requirements and the subtleties of the tasks' environment. This dissertation provides a framework of algorithms for automatic experience-based learning of such control tasks. To show the benefits of this framework, we focus on a number of well known problems that current learning algorithms suffer from and we propose various methods to overcome them.

1.1 Goals of the Dissertation

Typical source code of current state-of-the-art control programs for industrial applications contains innumerable hand-coded decision rules and parameters: Special cases are treated by *if*-statements. Parameters that define the control behavior are manually tuned based on human experience. For example, handling and extending a program with a big number of *if...then*-statements is likely to become incredibly complicated because the source code is long and its numerous branches are incomprehensible. Further, countless parameters have to be tuned in order to specify the desired behavior for all different situations that might occur. Oftentimes these parameters are *magic numbers*. Substantial expert knowledge is required to understand their impact on the machine's behavior. These parameters have to be tuned manually by human experts.

Preferably, a machine should receive a task as an input and then perform the task completely autonomous without any further intervention of the user. The machine itself has to acquire the skills needed to solve the task. Moreover, it must work reliably and be user-friendly and comfortable. Many machine control applications demand highly autonomous software agents that can cope with complex control problems. But in fact, the ideal notion of an agent that is capable of learning how to solve a complex task in an entirely self-directed manner is an allurement and an utopia altogether. The major goal of this dissertation is to provide methods that work in a largely automated fashion and make the learning process comfortable, transparent, and practicable on real machines. In our context, this means that the methods must be easy to implement and understand, and that they must work accurately, reliably, and in real-time.

1.2 Contributions of the Dissertation

The methods proposed in this dissertation **learn by induction from experience and interaction** with their environment. The learning process is performed *autonomously by the machine*. The user only needs to specify *how* and for *how long* the machine *acquires experience*. It is essentially our intention to reduce both the time and the intellectual resources required to produce machine control code.

The approaches introduced in this dissertation affect problems that occur in the practical application of autonomous machine control. Among the key contributions of this dissertation are:

- The development of high-performance autonomous machine control systems requires intensive experimentation in controllable, repeatable, and realistic settings. The need for experimentation is even higher in applications where the machine should automatically learn substantial parts of its behavior. The high accuracy of our proposed machine simulation based on neural learning allows for an efficient development of learned controller systems even in the case of a strongly limited availability of training data of the real machine.

- Unfortunately, typical learning methods for real world machine control tasks have a number of problems: It might be difficult to get a sufficient amount of training data for learning either because it takes too long or because no appropriate training data can be obtained from exploration at all. Furthermore, most current learning algorithms suffer from the incremental summation of errors during learning. In order to obtain a sufficient amount of training data we introduce adequate exploration techniques, such as backward exploration, and discuss how to acquire appropriate training data.

The non-incremental learning process described in this work avoids by design the incremental accumulation of errors during learning.

- Most machine control tasks have continuous state space. For the majority of tasks, the machine's response to control commands given a particular state is close to deterministic. But most current learning algorithms rely on a complex and often discrete MDP-model of the continuous state space. Operating in a discrete state space, the control output is not smooth and using a fine discretization the number of states becomes huge. The proposed methods require no discretization or any other kind of preprocessing of the machine's state space. In particular, no discrete MDP-model is needed. Our methods work reliably in continuous state spaces and continuous action spaces.

Furthermore, we propose ways to integrate a priori knowledge, such as undesirable states, in our learning algorithms. We choose suitable function approximators for the introduced learning methods and substantiate our choice. These approximators yield reliable controllers even if there are only very few training samples. In addition, we discuss how different layers of learning can be combined in case of hierarchical learning as well as in case of distributed learning agents.

Most of the approaches proposed in this dissertation have been developed in the context of robot soccer. Therefore, our learning methods depend on the physical behavior of real machines and are designed in order to run on real robots. Moreover, the proposed algorithms have been applied to real robot problems where they have succeeded under competitive conditions, such as robot tournaments. The algorithms have been extensively evaluated in simulation and on real machines to prove their reliability.

1.3 Outline of the Dissertation

In chapter 2 we present an introduction to control systems and the problems related to learned controllers. Chapter 3 delineates how we simulate machines in order to use the obtained simulators as a powerful tool in the development of control software. In chapter 4 we introduce the learning methods that we employ in our experience-based approach to control. Chapter 5 contains descriptions of the tasks that we successfully applied our approach to. Chapter 6 shows how layered experience-based control can be applied to highly complex tasks. In chapter 7 we discuss related work of both theoretical and practical nature and put it in relation to our methods. Finally, chapter 8 closes with a set of conclusions that we have drawn from this work. In the appendix, we describe function approximators and the specific control application *RoboCup* in greater detail.

Chapter 2

Control of Autonomous Mobile Systems

Generally speaking, the word *control* is used to describe a process of adaptation. In this process, one or more parameters represent the state of a separate system. These parameters are adapted in order to reach a desired value using a feedback loop based on measurements. According to this definition anything from a toaster oven to an autonomous space mission can be regarded as containing some level of control. In this thesis, we confine ourselves to the control of *autonomous mobile systems*. These systems are characterized as being able to autonomously guide mobile objects, e.g. mobile robots, auto-pilots, and grasping arms. All these applications become more and more important in our daily life.

In this chapter, we introduce some definitions for control and address and describe typical properties of real world control problems and what these properties mean for the development of reliable control software and, especially, for the development of control software using experience-based learning. For a general introduction to control theory [Doyle *et al.* 1992, Jacobs 1993, Franklin *et al.* 1994, Kuo 1995] are recommended.

2.1 Introduction to Control

2.1.1 Relevant Definitions for Control

Controlling and Steering In contrast to just steering a machine, control includes the existence of a closed *control-loop*. The control software must have knowledge of the parameters to control. These are called *controlled variables*. They are obtained from sensors directly or indirectly.

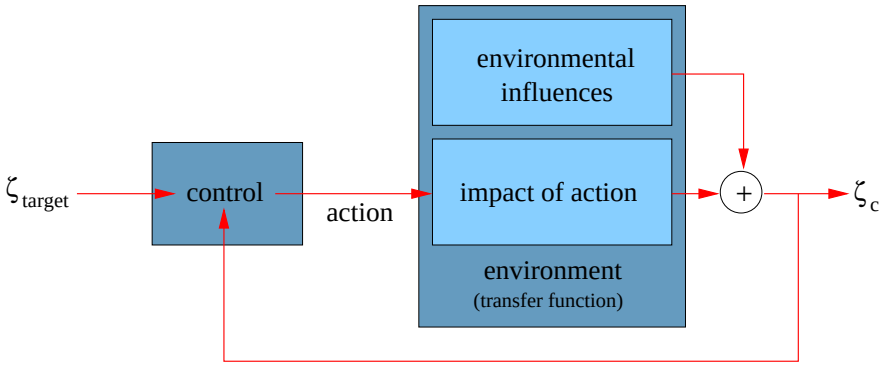


Figure 2.1: A simple control loop: The control unit performs an action given the current state ζ_c and a target state ζ_{target}. The impact of the action and environmental influences affect the current state ζ_c which is fed back to the control unit.

The following example illustrates the difference between steering and control. A driver of a car usually only *steers* the car by tuning *manipulating variables*. But if the driver keeps checking the speedometer in order to implement his intention of accelerating from say 60km/h to 100km/h he *controls* the car. In this case, the control-loop is closed since the *controlled variables* are known and there is a direct feedback.

System State and Actions In the context of control systems, we call the set of controlled variables the *state* (ζ) of a system. The tuning or modifying of the system state (using manipulating variables) is referred to as performing an *action* (a). Actions are chosen from a defined *action space* $\mathcal{A}$. The *control error* is defined as the distance between the measured controlled variables (start state) and the desired controlled variables (target state). The state of a system is described by a set of components, its *features*.

As an example consider a mobile object that is placed at a start state $\zeta_{start} = (0, 0)$ in a two-dimensional coordinate system with the Euclidean distance metric and a target state $\zeta_{target} = (1, 1)$. The control error, or distance between ζ_{start} and ζ_{target}, is $\sqrt{2}$. The features of the state space $\mathcal{S}$ are the coordinates of the system.

At times, at least one feature of the target state is not fixed to a specific value but can rather be selected from an interval or range of values. As a consequence, the target state is replaced by a *target space* ($\mathcal{S}_{target} \subset \mathcal{S}$).

While the majority of control tasks of practical interest have a one-dimensional state space ($dim(\mathcal{S}) = 1$), i.e. there is only one controlled variable, in this work we will focus on multi-dimensional control tasks. Multi-dimensional action spaces are considered as well.

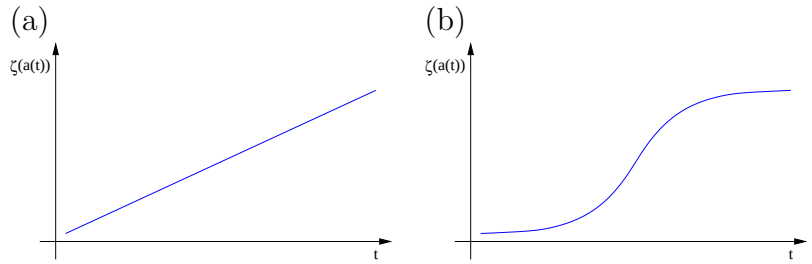


Figure 2.2: Two types of transfer functions: Subfigure (a): A direct (linear) transfer function. It is seldom found in real world control problems. Subfigure (b): A hysteresis. This nonlinear function is common in electromagnetic processes (e.g. motors).

Note, that the state space $\mathcal{S}$ and the action space $\mathcal{A}$ are not necessarily limited to real numbers. The action space could contain a number of possible algorithms for example. The state space could have colors as components.

The Basic Control Loop The process of control consists of a repeated generation of actions and a subsequent update of the current state. The development of the state depends on the impact of the action as well as on external factors of the environment that are beyond the influence of the control system. We call these external factors the *environmental influences*. Figure 2.1 illustrates a simple control loop. The state $\zeta(t + \Delta t)$ at time $t + \Delta t$ depends on the state $\zeta(t)$ at time t and the action $a(t)$ at time t and is given by

$$\zeta(t + \Delta t) = \zeta(t) + \Delta\zeta(\zeta(t), a(t), \Delta t) \tag{2.1}$$

where $\Delta\zeta$ is an update function that computes the change in state. If we assume Δt to be a unit of discretion in a discrete control process $\zeta(t + \Delta t)$ depends only on the previous state $\zeta(t)$ and the previous action $a(t)$. The new state is then computed by a projection-function $\mathcal{P}$:

$$\zeta(t + \Delta t) = \mathcal{P}(\zeta(t), a(t)) \tag{2.2}$$

2.1.2 Control Task Properties

Controllers differ widely in their properties concerning signal transmission, dead time, and oscillation. These aspects are described in the following.

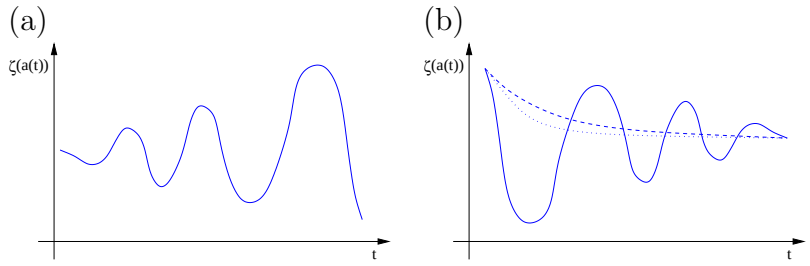


Figure 2.3: Oscillation: Subfigure (a): Increasing oscillation, the target state cannot be reached. Subfigure (b): Solid line: Decreasing oscillation, the target state is reached slowly. Dashed and dotted lines: The target state is reached without oscillation.

Transfer Function A basic component of every control loop is the transfer function. It defines the *behavior of the system*. Usually, the exact nature of this behavior is unknown and it contains the reaction of the environment to a certain state and a certain action. It may also contain environmental influences that include behavior that is out of the influence of the controller (see fig. 2.1). We distinguish between linear and nonlinear transfer functions. The term linear refers to the shape of the function, which maps from time to state, applying a constant action at every time interval (see fig. 2.2), and with it to the underlying differential equations. If, for example, a robot which is not moving receives actions to move forward we look at the changes in state over time, dependent on the time (and with it on the actions). If this curve is not linear we talk about a nonlinear transfer function. In real world control most transfer functions are nonlinear.

Oscillation In the majority of cases controlling a system towards a defined target state involves a tradeoff between time need and accuracy. A good controller quickly reaches the target state and maintains it stably. On the other hand, a high update rate in the controller always bears the risk of creating oscillation. In the worst case an increasing oscillation is created that finally results in a system collapse (see figure 2.3(a)). A system that shows a decreasing oscillation will ultimately reach the target state (see figure 2.3(b) solid line). A good controller reaches the target state without oscillation (2.3(b) dashed and dotted lines).

Changes in the Transfer Function The behavior of a transfer function may vary over time. This can be caused by situations such as a change in temperature or the wear and tear of mechanical parts. In these cases the control unit must be adaptive in order to be able to cope with the changes.

Dead Time Delays If the action chosen by the control unit affects the state of the system only after a certain time delay, this time delay is called *dead time*. No real world machine control loop works without dead time. The point is to keep it minimal. Excessive dead times lead to oscillation since actions are chosen at a time far away from the time that the actions influence the system.

Adversarial Environments In some cases, the transfer function not only depends on the action chosen by the control unit but on certain external factors, too. This influence is summarized under the term *environmental influences* in figure 2.1. Environmental influences can be an inherent behavior of the mechanics of the system at hand. But it can also be the result of another intelligent external component. Especially in competitive environments such a component can and will have a negative impact on the control process of a system. As an example consider two non-cooperating robots that want to manipulate the same object independently and in a different way. To cope with such a scenario we have to include information about independent external components in our state space.

Applications There are many real world problems in which controlling can be applied successfully. However, most of them have a one-dimensional state space. A simple example for this is a refrigerator which only has to keep a certain temperature. These kinds of applications are generally fairly straightforward to solve and do not call for sophisticated software solutions. From a scientific point of view, the more interesting tasks have a multi-dimensional state space and sometimes even a multi-dimensional action space. Tasks like auto-pilots, robot navigation, and cooperative multi-machine control belong into this category. It is this class of tasks that we want to apply experience-based learning techniques to.

2.1.3 Control and Experience-Based Learning

Learning

If we talk about learning in the context of control we first need to understand what learning strategies exist and which ones can be applied successfully. Learning itself has been defined in various ways. Some of the most popular definitions are provided in [Simon 1983, Michalski 1986]. In a general sense, we understand learning as the ability to apply knowledge gained from experience. The long term result of learning must be an increasing efficiency (in terms of well-defined criteria) in solving a particular task.

Deduction and Induction

Deduction is a process in which *no new knowledge* is gained. Instead, knowledge is only exploited for reformulation. For example the terms $b_1 \rightarrow b_2$ and $b_2 \rightarrow b_3$ can be combined to $b_1 \rightarrow b_3$ by deductive learning. In terms of control this means that deductive learning can only store experiences and apply them to the exact same situations that happened during the collection of the experiences. In any other situation a deductive controller will fail.

Induction is the generalization of knowledge obtained from experience. For example the term $b_1 \rightarrow b_2$ could be generalized to $\tilde{b_1} \rightarrow b_2$ if $\tilde{b_1}$ is similar to b_1. A good induction might even generalize to $\tilde{b_1} \rightarrow \tilde{b_2}$ with $\tilde{b_2}$ being similar to and more appropriate than b_2. This ability makes inductive learning a very promising method for the learning of a controller.

Supervised and Unsupervised Learning

Equipped with our understanding of the notion of learning the question remains of how to actually perform learning. The most common approach is *supervised* learning. In supervised learning a usually human *teacher* produces training data (such as a rule $b_1 \rightarrow b_2$) and provides it for learning. In control, training data can be obtained automatically from randomly steering for example. Common representatives of supervised learning are multi layer perceptrons, decision trees, and CMACs. These are all described in the appendix.

If a system uses *unsupervised* learning it must be capable of organizing its knowledge autonomously. This could mean that the rule $b_1 \rightarrow b_2$ is created out of the data points b_1 and b_2 only. In general, unsupervised learning is applicable to control problems but requires sophisticated data organization algorithms. Popular methods of unsupervised learning are Kohonen networks [Kohonen 1988] as well as various clustering algorithms [Duda and Hart 1973].

Besides supervised and unsupervised learning there are some techniques that do not fit in either of these schemes. Reinforcement learning (see section 7.1), to name a major method, applies supervised learning to data that before has been acquired autonomously by only rewarding the algorithm based on the quality of its actions. So there is a supervised component to reinforcement learning that can be classified as *indirect supervised learning*.

2.2 Problems of Learned Controllers

If we want to use direct or indirect supervised learning to solve control tasks, we have to think about a variety of issues. These include the representation and

acquisition of training data, the structure of the state space and the action space, and the approximation of the functions we employ for learning.

2.2.1 Training Data

Both, the quality and the quantity of the training data at hand determine to a great extent if the system performs well given specific control task. In the following the most important things to consider about training data are discussed.

Representation

In the context of this work, we assume the training data to be a set of patterns that can be used to learn a function $f : \mathbb{R}^d \rightarrow \mathbb{R}^n$ that defines a mapping from a d-dimensional input space to an n-dimensional output space.

To approach a new control task, we first have to define the input space and the associated output space. In practice this means nothing else but to describe states and actions with numbers. In many cases, real numbers are a good choice but in certain cases integer numbers can be more appropriate. Note that the number of dimensions we need to describe the state space and the action spaces *can* provide information about the complexity of the learning task.

Reduction of the state space

Before generating learning data it is advisable to think if there might be redundancy in the data. For example consider a machine that when given action a in state ζ_0 reaches state ζ, and that when given action $-a$ in state ζ_0 it reaches state $-\zeta$. To learn this behavior we only need a semi-space for learning. Along similar lines, there might be symmetries in points or planes which can allow for a significant reduction of the input space. Furthermore, an action a in state ζ might have the same impact as an action $a+\Delta a$ in state $\zeta+\Delta\zeta$ where the relation between Δa and $\Delta\zeta$ is known in advance. In this case learning can be simplified substantially by only learning the relation between a and ζ.

Scaling and Transformation

Usually, each dimension of the state space (and action space) has a particular interval where most of the data is concentrated. In fact, typically only a limited interval of the real numbers $\mathbb{R}$ is suitable to characterize a state or an action at all. For example the temperature of a refrigerator will never be outside a limited interval, say 0C and 10C, during normal use. Along the same lines, the velocity

of a robot is limited by variables such as the power of the robot's motor, friction, and others. Hence the robot's velocity will not exceed a certain value.

If the input space has more than one dimension it is possible and likely that the data is scattered over different regions of $\mathbb{R}$. For example the data of a two-dimensional input may be distributed over $[0.7, 0.8] \times [-10000, +10000]$. It can be difficult for certain learning algorithms to deal with these differences in scale. This can be especially true if small changes in a small interval have a greater impact than big changes in a big interval. As a consequence, the data is generally normalized by scaling the relevant interval of every dimension to the same particular interval across all dimensions (usually $[0, 1]$).

Furthermore, it is possible that the data of a dimension of the input space is of highly nonlinear nature. Consider an input interval $[0, 100]$. It is possible that the difference between 0 and 10 has no noticeable impact on the output but that the difference between 50 and 51 accounts for a significant change. This behavior can be observed, for example, by the response of an electric motor to changes in the voltage of the input current. In such cases it has proven superior to scale the data using nonlinear functions (e.g. logarithm, sigmoid, square root, and polynoms).

It is problematic to use features of periodic nature, such as angles, as dimensions of the state space. Depending on the application, an angle of 360 degrees might have the same meaning as an angle of zero degrees. However, learning algorithms are not readily aware of this fact and might be presented with two learning patterns with nearly the same input but an entirely different output (0 and 360, respectively). In our experiments we avoid periodic features by transforming them into other, non-periodic features or by computing relative deviations instead.

Finally we want to address the problem of unbounded numbers in the output space. Learning algorithms like neural networks inherently make use of functions that map to $[0, 1]$ so they work with output spaces $[0, 1]^n$ only.

Data Acquisition

The acquisition of training data is an important part of the learning process: Both, the amount and the quality of the data at hand have a significant impact on the success of the learning. A general rule says: *The more training data, the better the result.* This holds true as long as the data meets certain preconditions. If the available training data does not (more or less) uniformly cover the valid input space it may be difficult for the approximator to generalize to input samples that lie in areas of the input space for which no training patterns are provided regardless of how much data is available (see figure 2.4(a)). Therefore it is essential to cover the state space as uniformly as possible with training data (see figure 2.4(b)).

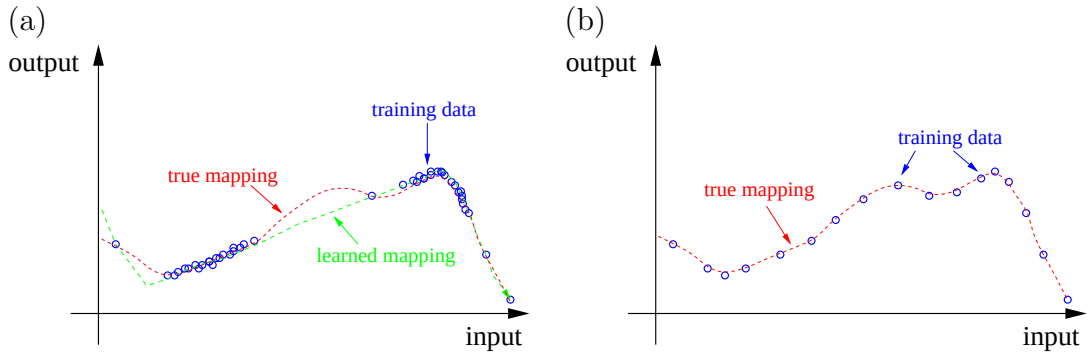


Figure 2.4: Subfigure (a): The training data covers only a small part of the input space. As a consequence a partially wrong mapping might be learned. Subfigure (b): The input space is uniformly covered by training data. The learned mapping is more reliable.

Noise In machine control tasks the training data is usually obtained from sensors. Due to noise in every sensor, its measurements are typically not 100 percent accurate. We have to design our learning algorithm such that it is robust enough to deal with this inaccuracy. As long as the noise can be kept moderate it is fair to learn the noise with the learning function. Also, if the type of sensory error is known in advance the data can be preprocessed to eliminate or minimize this error. If a very noisy sensor is used to obtain learning data from exploration and if this compromised learning data, in turn, is used for further exploration there is a significant risk that the error becomes magnified. Over time, this may lead to an error that can decrease the performance of the controller substantially or, worse yet, make it completely useless. In sections 4.4.3 to 4.4.6 we explain how our approach to experience-based control manages to avoid the magnification of such errors.

Lack of Training Data For complex high-dimensional learning tasks vast amounts of training data are required but in practice it is a laborious task to collect and prepare a sufficient quantity of data. One way to overcome this is to construct an induction of the basic behavior of the machine. This will give us the opportunity to generate training data offline. One such technique is described in chapter 3.

Unsuccessful Exploration Some control tasks demand great skills. The simple example of a computer game shows that some tasks cannot even be performed by inexperienced humans. For the experience-based learning of control software this brings a fundamental problem to the surface: What if there is no learning data that tells us how to get to a particular target state? How can we train a controller to reach a target state if it has never been in that state before? How do

we determine whether an action or a state is good or bad if we do not know how to reach the target from there? Up to now, these questions, have been largely neglected by the research that has been conducted on learning in control systems. By exploiting knowledge about the relation between states and actions we can overcome this problem as shown in section 4.3.2.

A priori Knowledge

In some applications a priori knowledge about the transfer function may be available. It makes sense to exploit this knowledge if there are indications that it will help to improve the performance of the system. If, for instance, a machine gets hot it might become necessary to suspend it for a period of time to avoid the risk of a fire. Dependent on the application, the handling of other exceptions might be appropriate. To reflect this knowledge in the training data patterns that describe such special cases could be constructed. Another possibility is to keep these special cases out of the learned controller altogether and instead employ a competent hand-coded decision module for these cases alongside with the learned controller.

Undesirable States In many practical applications not only do we want to reach a certain target state but we also want to avoid certain undesirable (e.g. potentially harmful) states. For example, we might want for a mobile robot to avoid areas of strong magnetic fields as these might have a noticeable negative impact on the system. To handle such cases we need to be able to model them explicitly in the controller. So far, most learned controllers do not provide this capability. In section 4.4.4 we describe how we managed to add this capability to our learned controller.

2.2.2 Discrete and Continuous State Space

Most state spaces of real world problems can be described sufficiently by a set of continuous features. This implies that the system can assume an unlimited number of states. But some learning algorithms rely on the assumption that the state space is discrete, i.e. that there is only a finite number of states that the system can assume. For example the basic model of a finite Markov Decision Process [Ross 1983, Bertsekas 1987] assumes discrete states with discrete actions as transitions between them.

Practical state spaces can include features such as distances, angles, and the velocities of machines or parts thereof, and other objects. Since acceleration, in general, is nonlinear velocity values will be hard to be discretized reasonably. In a high-dimensional state space that includes nonlinearities, determining the set

of states that can be reached from a given discrete state with one action becomes virtually impossible.

Further discretization may lead to extremely discontinuous behavior. This is especially true in cases where small deviations in the input can cause substantial deviations in the output. So, when a coarse discretization is applied, the control output is not smooth, when a fine discretization is used, the number of states becomes unmanageably large, especially in high dimensional state spaces [Doya 2000].

2.2.3 Function Approximation

The actual process of learning consists of the approximation of an unknown function $f : \mathbb{R}^d \rightarrow \mathbb{R}^n$ that maps from a d-dimensional input space to an n-dimensional output space. In the simplest form, the learning data can be represented by storing it verbatim in a discrete lookup table. This is transparent and without loss of any original training information. And it is efficient up to a certain degree of complexity of the approximated function. If the input space, and as such the learning data, becomes very high-dimensional it will become unmanageable to represent all learning patterns explicitly in a lookup table. The patterns have to be learned by a function approximator.

Models of Functions

If mathematical characteristics of the function to be approximated are known a parameterized model of it can be developed. Then only the parameters of this model need to be learned. A very simple model is the linear function $f(x) = b_1 x + b_2$. Other common models are polynoms $f(x) = \sum_{i=0}^{n} b_i x^i$ or functions of the form $f(x) = b_1 \cdot sin(b_2 x)$. All of these models have the drawback that they can only be used for applications that they are suitable for and that their modeling complexity is limited by their model.

Approximators

Unlike lookup tables *nearest neighbor* algorithms can deal with continuous output spaces. Patterns are stored efficiently and the approximator output is computed by the weighted output of the input's nearest neighbor training patterns (see appendix A.4 for details).

CMACs (Cerebellar Model Articulation Controllers, see appendix A.1 for details) consist of receptive fields that are responsible for certain regions of the input space and are often used for function approximation in reinforcement learning. But they are not without drawbacks, (1) the resulting function is not continuous in its

Function Approximator	Rule Transparency	Runtime (Training)	Runtime (Application)	Type of Input/Output	Modeling Power
Lookup Tables	good	good	satisfactory	continuous/ discrete	any(*)
Model of Function	satisfactory	good	good	continuous/ continuous	limited
Nearest Neighbor	satisfactory	satisfactory	satisfactory	continuous/ continuous	any(*)
CMAC	satisfactory	satisfactory	satisfactory	continuous/ discrete	any(*)
RBF Network	satisfactory	satisfactory	satisfactory	continuous/ continuous	any
Multi Layer Perceptron	poor	poor	good	continuous/ continuous	any
Decision Tree	good	satisfactory	good	continuous/ discrete	any(*)

Table 2.1: Function approximators and their characteristics. (*) Any level of complexity can be modeled but the required amount of parameters for complex functions can be very high.

output and (2) the number of receptive fields becomes huge when approximating complex functions with high-dimensional input spaces.

Networks of Radial Basis Functions (RBF-Networks) also consist of units that are responsible for certain regions of the input space. Their basic units are Gaussian functions (see appendix A.3 for details). The output function is continuous and the number of Gaussian functions can be kept manageably small as long as the learning patterns with similar input also have similar output.

Multi Layer Perceptrons (see appendix A.2 for details) are neural networks based on the basic Perceptron [Rosenblatt 1957, Rosenblatt 1958, Rosenblatt 1962]. Their high level of structural complexity makes them very powerful but also very intransparent. Their computational complexity for training tends to be high but the amount of computation required to determine the output for a given input is very low. MLPs have the potential to approximate functions of arbitrary complexity.

Decision Trees contain the output in their leaves while the input is used as a means to find a corresponding leaf. At each inner node a choice between a finite number of paths has to be made. That means that decision trees (see appendix A.5 for details) are suitable for discrete output spaces. Furthermore, they are very transparent since they are based on rules of the form $\mathsf{if} \wedge_{i=0}^{\#rules} c_i \ \mathsf{then}\ o = \ldots$

where c_i are conditions at inner nodes, $\#rules$ is the number of rules, and o is the output of the tree.

In appendix A the most common function approximators are described in further detail. The software implementations of the function approximators that we used in our experiments is introduced in section 5.8.

Incremental and Non-Incremental Learning Function approximation can be implemented as an online or as an offline process. Approximating online means that the approximator continues to learn individual training patterns incrementally. In offline learning a set of training patterns is regarded as one block of training data and it is learned at the same time (at least from the user's point of view). Both methods, online and offline learning, have their respective advantages and disadvantages: Online learning is able to keep adapting to changes in the environment lifelong. On the other hand, the impact of old patterns (i.e. patterns that were learned a long time ago) might converge to zero. In contrast, offline learning guarantees that all information gained from learning is retained but the system is not able to adapt to any environmental changes. In general, a system that uses offline learning has two modes of operation: (1) learning and (2) exploitation.

For the most part, incremental learning techniques are more laborious to implement and only used if necessary. And since most machines act in state spaces whose dynamics are encoded in the state itself and do not change significantly over time, offline learning is suitable for a large variety of tasks. This will also be substantiated by our experiments in chapter 5. However in practice, combinations of online and offline learning techniques are also used in order to make an offline trained system more adaptive. Obviously, this approach is even more complex than any of the single methods it employs.

2.2.4 Discrete and Continuous Action Space

The action space of a control system can be discrete or continuous. When a discrete action space is also finite and it has a moderate number of actions, an exhaustive search over all possible actions can be performed to find the globally optimal action, i.e. the action that leads to the best successor state (according to some criterion). However, practical action spaces are typically too complex for such a search to be feasible. In these cases heuristics are needed that allow for an informed and nondestructive reduction of the search space to a size that an exhaustive search can be performed on in practice. In our context nondestructive means that the reduction will very likely not eliminate the best or at least most of the very good actions from the search. For example, first a search using a coarse discretization can be performed to determine a preliminary optimal action. Then,

using a fine discretization, a second search can be performed in the vicinity of that action to find a better and locally optimal action.

In practice, machines are generally controlled by continuous manipulating variables rather than discrete ones. As a consequence, a continuous action space is most often a more appropriate choice for a control system than a discrete action space. However, in systems with a continuous action space the number of possible actions for any given system state is infinite. Hence, again, heuristics need to be applied to find the action promising the best successor state in a reasonable amount of time. The only way to avoid the use of heuristics is by learning a direct mapping from a state to an action rather than by learning the impact of an action and the appropriateness of the resulting state. We will introduce this approach in section 4.4.6.

2.2.5 Control Tasks of High Complexity

There are control tasks that have such a high-dimensional state space that learning in this space is possible in theory, but in practice it is not feasible. Consider, for instance, the task of controlling a team of eleven soccer players. The dimension of this state space is at least 46 (i.e. two teams with eleven players each and a ball with their respective positions in 2D) and the action space has at least 11 dimensions (one-dimensional action space for each player to be controlled). To achieve a reasonable level of learning an unimaginable amount of learning patterns would be required. The resulting computational complexity is too high even for today's powerful computers. Even chess with its finite and discrete state and action space has so far not been played well with learned software. The state of the art method to master such high-complexity tasks is to use a divide-and-conquer approach. In it the learning task is divided into a hierarchy of smaller, more manageable tasks, each of which is then solved by itself. We call this method *layered learning* and describe it in greater detail in chapter 6.

Chapter 3

Simulation of Dynamic Processes as a Development Tool for Machine Control Systems

3.1 Introduction

The development of high-performance machine control software requires extensive controllable and repeatable experimentation in the target environment of the control system or a realistic approximation of it. If a machine is required to learn substantial parts of its control system autonomously the demand for experimentation is even greater.

Simulation is a very powerful tool in the development of intelligent machine control systems as it enables the developer to perform countless controllable and repeatable experiments and to make fast and inexpensive predictions. In fact, experiments can be run safely in faster than real-time using time lapse. Simulation eliminates the risk of hardware damage and can potentially reduce the cost for hardware and other materials significantly. Furthermore simulation allows for efficient diagnosis and correction of software errors through monitoring and stepping. However, the simulation environment must provide a close enough approximation of the actual target environment to warrant that the control software will operate flawlessly in the real system.

In this chapter, we propose and describe such a simulation environment and provide empirical evidence that it can in fact serve as a powerful tool in the development of even complex control modules for real robots. The control loop described in chapter 2, figure 2.1 is extended by a simulation component (see fig. 3.1).

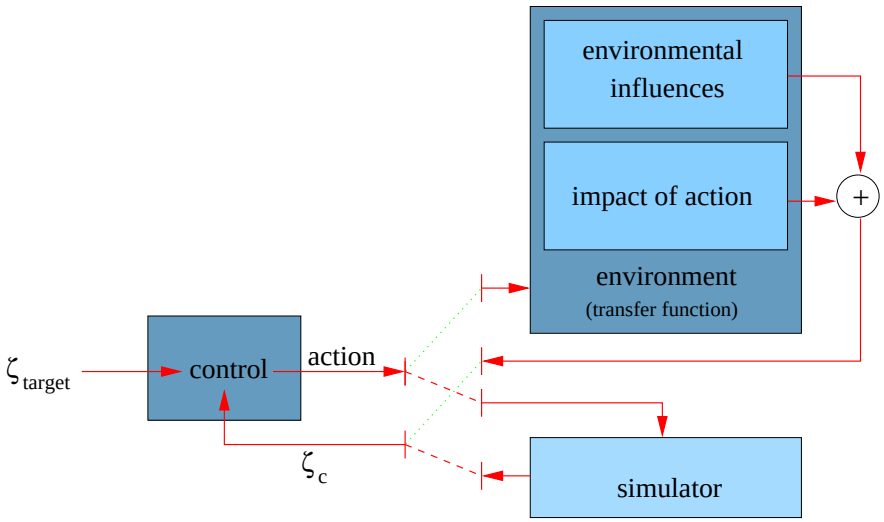


Figure 3.1: A control loop with a simulator: The control unit is tested by sending its actions to the simulator. Once the controller works reliably it can be applied to its real world environment.

3.2 Simulation as an Induction

In section 2.2.1 we stated that the acquisition of training data is an important precondition for the learning process and that we want to generate as much training data as possible. In practice, the quantity of available training data is a (roughly linear) function of time. Since time is limited, especially in the development of real-time control software, we need to find ways to acquire large amounts of training data quickly and economically. Let us consider the task of data collection through experimentation for the training of a real multi robot control system. Let the duration of an experiment with a real robot be two minutes (excluding the recharging of batteries, maintenance, etc). Given a simulator that is capable of mimicking the exact behavior of this robot, the experiments could be performed automatically, each of them started from a randomly selected initial state. If this simulator completes two experiments per second (which allows for quite complex tasks) the speed-up would be $24,000\%$.

For two reasons, this example does not provide an accurate estimate: (1) It assumes an exact simulator. (2) It neglects the time needed to develop the simulation. While a sufficiently accurate simulation is possible for most practical applications requires a more thorough discussion:

If the given control task has a high-dimensional state space that includes features from the robot's environment that can be set artificially in the simulation then the use of a simulator is very likely beneficial. Using field training data from a

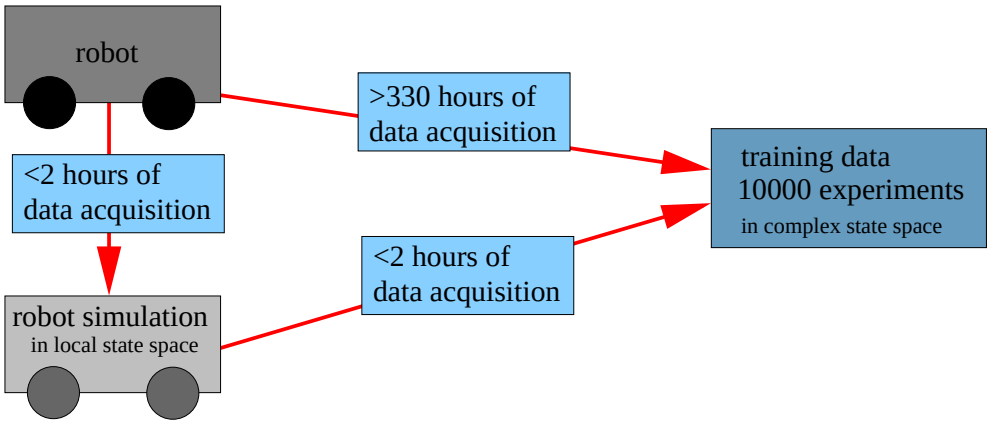


Figure 3.2: A sample learning task: Acquiring data needed to learn a robot's dynamics takes roughly two hours. Acquiring data required to learn the task in the simulator takes about two hours. In comparison, learning the same task in the real world without the simulator requires on the order of 330 hours.

small local state space of the robot (e.g. only the robot's velocity) an induction can be performed and used for the simulation of experiments in a much more complex state space. For instance, to determine how long it takes for a robot to reach a certain target state starting from a particular start state, we only need to learn the impact of the robot's control commands (actions) in certain states. Then, all expensive experiments that need to be performed to learn the task can be done in simulation. Using *temporally inexpensive training data from a lower level* we can do high-level experiments in simulation.

As a result, we propose to solve such learning tasks in a three step process. First, we learn a simulation of the machine's basic dynamics. Second, we perform the high-level learning tasks using this simulator. And third, we port the resulting controller onto the actual machine and cross-validate the performance gains obtained from the new controller. For example, we can use this method to determine how long it takes for a robot to reach a particular target state given a particular start state. We collect a mere two hours of training data using the real robot's dynamics and then perform the learning task in another two hours through 10,000 unsupervised simulator experiments (see figure 3.2). Assuming that setting up and executing a task takes only two minutes for the real robot, we would have had to spend more than 330 hours of experimentation with a real robot (not including recharging of battery, maintenance, sleeping and any other breaks). Obviously, this is not feasible.

A major precondition that the proposed method will work is the accuracy of the simulation. Although we will not be able to construct a perfect imitation of the

machine's behavior we can reach a certain degree of accuracy that suffices for most applications.

3.3 Simulation of Dynamical Properties

In addition to an accurate modeling of the target machine's sensors and its motion a simulator should be able to simulate different kinds of machines with their respective motion profile acting at the same time. A model of the machine's static environment as well as models of dynamic objects should be easy to integrate. Furthermore it is essential that a high number of learning data can be obtained in a reasonable period of time. This constraint forces us to use computationally inexpensive methods for the simulation. In general there are two different basic concepts for simulation:

White-Box Simulation If we know how to hand-code the simulation of the machine's dynamics or we can understand how a learned simulator works in detail we talk about a white-box simulation. In this case we are able to change the simulator's behavior in order to achieve improvements.

Black-Box Simulation If we do not know anything about the details of the simulation and are not able to predict what impact a change in the code of the simulator will have we talk about a black-box simulation.

In practice there are many simulators that are partly white-box and partly black-box. The parts of a system where the background is well understood by humans is put in a white-box simulation while the remaining parts (mostly impossible to be hand-coded) are implemented by learning algorithms, which are usually black-box and will not be edited directly.

3.3.1 Analytic Simulation of Dynamic Processes

The analytic simulation requires knowledge about the machine to be simulated in terms of laws of physics. Obviously, to provide a realistic simulation from a physics point of view we have to include a gigantic number of laws: Many factors such as temperature, atmospheric humidity, atmospheric pressure, magnetic fields, gravitation, luminous intensity, radiation of other frequency etc. *might* have influence on the machine. It would be absurd to construct a simulator that contains all physical properties of the real world. Therefore it is absolutely necessary to know which factors have a major impact on the behavior of a machine.

In many cases, fortunately, it is sufficient to take the most important factors into account to obtain a simulation of high quality.

Let us assume we have a mobile robot that we can give target positions p_{target} to go to. Let us further assume that we know that this robot will directly move there and can reach a maximal velocity of V_{max}. In this case a very simple simulation would be to compute the current position $p(t + \Delta t)$ of the robot for each discrete time interval of Δt by

$$p(t+\Delta t) = \begin{cases} p_{target} & if \quad |p_{target} - p(t)| \leq \Delta t \cdot V_{max} \\ p_{target} \cdot \frac{\Delta t \cdot V_{max}}{|p_{target} - p(t)|} + p(t) \cdot (1 - \frac{\Delta t \cdot V_{max}}{|p_{target} - p(t)|}) & else \end{cases} \tag{3.1}$$

Obviously, this model inherently neglects acceleration. A correct physical simulation computes the forces affecting the movement (and with it the velocity) of an object:

$$V(t+\Delta t) = V(t) + \frac{\sum_i F_i \cdot \Delta t}{m} \tag{3.2}$$

F_i are the forces affecting the movement of the respective object. As previously mentioned, it is necessary to know all relevant forces. Others that are of a low absolute value can be neglected. m is the mass of the object regarded. The new state of the object after a discrete time step Δt is computed by

$$\zeta(t+\Delta t) = \zeta(t) + V(t+\Delta t) \cdot \Delta t \tag{3.3}$$

The degree of discretization is selectable while a fine discretization needs a higher amount of computational resources and a coarse discretization will be less precise.

The main advantage of an analytic simulation is its white-box character and the fact that it relies on laws of physics that have been proven to be correct. On the other hand the disadvantages are that all relevant forces F_i must be known (completeness of the model) and constants and parameters (for instance a c_w-value) must be set as well.

3.3.2 Neural Simulation of Dynamic Processes

A neural simulation of a dynamic real world process requires the opportunity to record data that results from interaction of states and actions. This training data is essential for the neural learning of the dynamical behavior. In contrast to an analytic simulation we do not have to care about the physical background of the process. We need no knowledge on the relevance of forces, rules, constants or any kind of parameters.

The information to be recorded consists of two data streams: (1) The sequence of control commands (actions) that are sent to the machine. (2) We need to know as much as possible about the changes in the state of the machine. All these pieces of information are essential for learning a relation between states and actions that can describe the dynamical behavior of the machine.

Sequences of States and Actions The actions of a machine can easily be recorded by just writing them in a file using time stamps. The result will be a sequence of actions $\bar{\mathcal{A}}$

$$\bar{\mathcal{A}} = \langle a(t_0), a(t_1), a(t_2), ... \rangle \tag{3.4}$$

where $a(t_i)$ is the command (action) sent to the robot at time t_i. Similarly, we need a sequence of the machine's states $\bar{\mathcal{S}}$

$$\bar{\mathcal{S}} = \langle \zeta(t_0), \zeta(t_1), \zeta(t_2), ... \rangle \tag{3.5}$$

where $\zeta(t_i)$ is the state of the machine measured at time t_i. While actions are easy to obtain states must be measured by sensors but sensors are noisy. So if we construct a simulation we must be aware of the fact that it relies on some sensors that *might* cause errors. To find out to what extent the given sensors are noisy we can compare the results of different sensors or even compare them to measurements done by hand. As long as sensors provide accurate data in between short intervals of time they are good enough to be used for recording states for simulation. An error that results from the summation of a number of small and negligible errors over time is not problematic. Such an error can be balanced by a high-level feedback control loop.

Time Delay and Synchronization Since we use a relation between states and actions for simulation we must have a look at which control commands actually have an impact on the current change in state of the respective machine. For example, if we have a hardware system with a high dead time delay it is probably not the last action executed that has the most impact on the next state measured. To support the understanding of the temporal relationship between actions and affected states it is extremely helpful if we can rely on

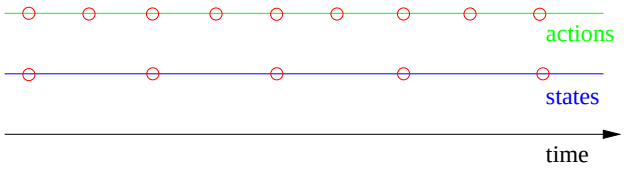


Figure 3.3: Executing actions and measuring states should be done synchronously to support the understanding of the relation between actions and states.

synchronous sequences of states and actions, both with a fixed cycle-time (see figure 3.3).

Learning a Relation between States and Actions Once we have all the data necessary to learn a relation between states and actions (the sequences introduced above) we need to generate patterns for learning from this data. Let us define the cycle-time of the recording of the actions Δt_a and the cycle-time of the measurements of the states Δt_ζ. Further, let t_{la} be the time where the last action was executed. To learn which actions have an impact on the change in state from $\zeta(t)$ to $\zeta(t + \Delta t_\zeta)$ we generate patterns

$$\langle \zeta(t), \langle a(t_{la}), a(t_{la} - \Delta t_a), a(t_{la} - 2\Delta t_a), a(t_{la} - 3\Delta t_a), ... \rangle \rangle, \zeta(t + \Delta t_\zeta) \tag{3.6}$$

These patterns are used to learn a mapping

$$\mathcal{P} = \begin{cases} \mathcal{S} \times \bar{\mathcal{A}} \to \mathcal{S} \\ \zeta(t), \langle a(t_{la}), a(t_{la} - \Delta t_a), ... \rangle \mapsto \zeta(t + \Delta t_\zeta) \end{cases} \tag{3.7}$$

that maps from a state at time t, $\zeta(t)$ and a sequence of past actions to the succeeding state at time $t + \Delta t_\zeta$, $\zeta(t + \Delta t_\zeta)$. We regard only the information of the current state, $\zeta(t)$. With it, we formally assume the Markov property. In fact, we consider further information by regarding the sequence of actions, $\bar{\mathcal{A}}$. These actions may have an impact on previous states. The sequence of the last actions is chosen dependent on the application, its dead time behavior, and the cycle-times. In cases that are optimal from the developer's point of view the sequence of the last actions can be shrunk to at least one action. The complexity of the learning task, inherently, depends on the size of the sequence of the last actions. If we do not know how to choose this sequence it is advantageous to first choose a bigger number of past actions rather than a number too small.

The patterns generated from recorded data (equation (3.6)) can be learned by function approximators (see appendix A). We propose to learn them using multi layer neural networks (see appendix A.2). The neural simulation, naturally, is a black-box simulation since we do not know what exactly happens inside the neural networks. The big advantage of this simulation is that we do not have to know anything about the physical background of the simulated machine. Neural networks can learn even highly nonlinear functions. The drawbacks are that we cannot adapt the simulation because of its black-box character and that for different discretizations (different cycle-times) we have to train different neural networks.

3.4 Simulation of Sensory Properties

Besides the simulation of dynamical properties we have to think about the simulation of sensory properties as well. Similar to the simulation of the machine's dynamics we have to build a model for another part of the machine's hardware. There are a number of "easy" methods that typically achieve only a limited degree of accuracy; on the other hand a precise simulation of sensors (especially cameras) is a laborious task.

The Truth-Assumption A trivial way to solve the problem of the simulation of sensory properties is for the machine to receive its current "true" state as it is computed by the simulation of the dynamical properties. Doing so we assume that the sensors of our robots are near perfect which obviously is a wrong assumption. But this approach works well with accurate sensors. The higher the accuracy of the sensors the better are the results achieved with this approach. There is a great chance that the control software developed using this approach will work with accurate sensors.

Gaussian Fuzzifying An extension of the truth-assumption is to have the machine receive its current state data (as it is computed by the simulation of the dynamical properties) with some amount of simulated noise. Using Gaussian functions for the computation of the noise we cover a lot of possible sensory errors that may occur in practice. Furthermore, we can integrate knowledge about sensory errors of the respective application because a Gaussian fuzzifier is a white-box simulation. For instance, a typical behavior of visual localization is that distant features are located with an error much higher than features that are close to the camera. Typically, the error in the distance of a feature is greater than the error in the angle. This knowledge can be integrated in Gaussian fuzzifying by using multi-dimensional Gaussian functions to simulate the noise (see figure 3.4 for an example).

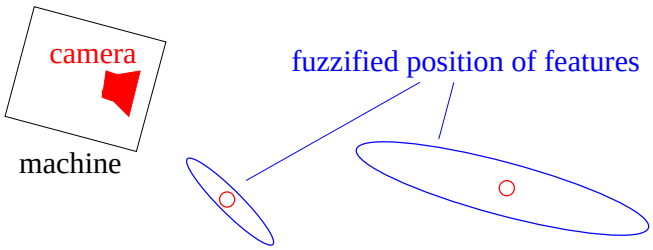


Figure 3.4: Gaussian fuzzifying. Features close to the camera of the machine will receive less noise than distant features. The angle will be less noisy than the distance.

Learning the Simulation of Sensory Properties The methods described above both make assumptions about the sensors of a machine. Since these assumptions usually are not met in practice we have to work on a more realistic approach. To achieve a realistic simulation of the sensors we must learn the mapping

$$\mathcal{I} : \zeta(t) \mapsto \Upsilon(t) \tag{3.8}$$

from a machine's state $\zeta(t)$ at time t in the simulator to the sensory data vector at time t, $\Upsilon(t)$. This task brings along some major problems. The first issue is where to get training data for $\mathcal{I}$ from. In the real world we can measure the vector of sensory data $\Upsilon(t)$ but how can we reliably measure the "*true*" state of the machine, $\zeta(t)$? In terms of a mobile robot we can record the "*true*" state of the robot, $\zeta(t)$ by a camera mounted at the ceiling of the room and record the sensory data vector, $\Upsilon(t)$ in parallel. This solution would only include errors of the camera mounted at the ceiling. These errors can be kept moderate. But cameras, in general, depend on factors like the illumination of the environment. So another major problem is the simulation of visual sensors under different lighting conditions. Data obtained from the recordings of $\zeta(t)$ and $\Upsilon(t)$ can be learned by using a function approximator (see appendix A for details on function approximation).

3.5 Multi Robot Simulation

Robot simulation, as machine simulation in general, is a powerful tool for the development of intelligent control software because it allows for fast and cheap predictions and makes experiments controllable, repeatable, and scalable without the danger of damaging hardware.

However, the use of a robot simulator also bears a number of problems. For the purpose of simulation, typically time has to be discretized. Also, simulators normally work in an abstract feature space and might therefore ignore key factors for the real robot behavior [Lee *et al.* 1999]. Others argue that simulated controllers are doomed to succeed because of the design of the simulators [Brooks and Mataric 1993]. As a consequence, software that succeeds in simulation may fail on a real robot [Brooks 1992]. Accurate and analytic simulations can typically become extremely complex and expensive in terms of computational resources (see section 3.3.1). Thus they are often performed in parallel and distributed simulations [Jugel and Sydow 1998, Mehlhaus and Rausch 1993]. Despite these problems a number of robot simulators have proven to be a valuable resource in the development of robot controllers [Jakobi *et al.* 1995, Buck *et al.* 2002c].

(a) (b)

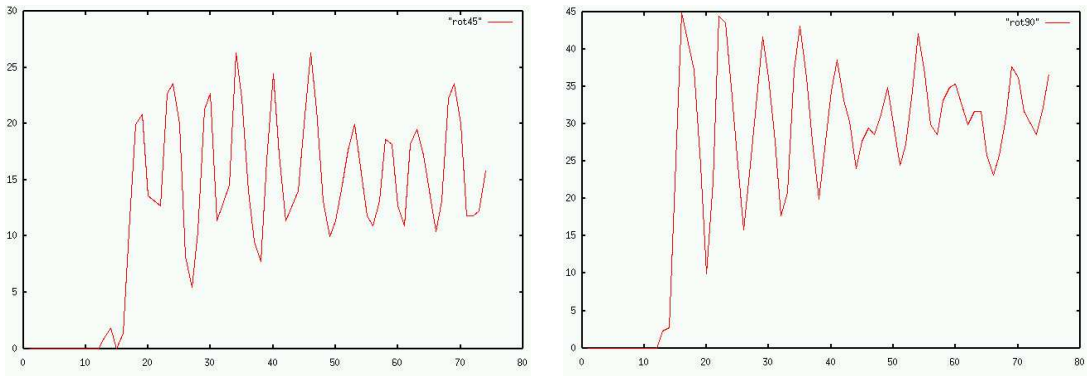


Figure 3.5: The development of the rotational velocity (degrees per second) of a Pioneer I robot over time ($\frac{1}{10}s$). The robot starts at $V_{rot} = 0$. Subfigure (a): The robot is constantly given the control command $a = (0, 45)$. Subfigure (b): The robot is constantly given the control command $a = (0, 90)$.

Multi robot simulation includes the simulation of sensing as well as the simulation of motion dynamics. The neural simulation of motion maps the dynamic state of the robot and a sequence of low level robot commands into a successor state (see equation (3.7), section 3.3.2). Sensor simulation (see section 3.4) maps the local surroundings and the sensor model into the sensed data. A substantial amount of research work has been done on either one or both of these aspects: [Lee *et al.* 1998] generates an artificial neural network based model of the environment within a simulator of a Nomad robot, to learn an action model in an MDP framework simulators have been used [Belker and Beetz 2001], and many experiments of the successful well known rhino robot [Thrun *et al.* 1998] were done in simulation. Furthermore the soccerserver [Kitano *et al.* 1997, Noda *et al.* 1998] used in the RoboCup Simulation League mimics some sensory and motion abilities of a human-like soccer player. Another RoboCup simulator [Kobialka *et al.* 2000] is able to simulate a large number of different sensors (infrared, bumper, camera, and laser) while motion is simulated by using the values of translational and rotational control commands directly in order to compute a succeeding state.

While most of the above work concentrates on the simulation of sensors by more or less neglecting the simulation of motion, our main goal in this section is to construct an accurate simulation of the robot's dynamical behavior. This becomes very important in high-speed environments such as autonomous robot soccer or automated packing applications. In robot soccer abrupt changes in speed and direction are as common as they are in a real soccer game. We will apply the neural learning from real robot data as described in section 3.3.2 and show that it is a highly qualified approach because of its high accuracy.

3.5.1 Neural Simulation of the Dynamical Properties of a Pioneer I Robot

In this section we describe how to learn a dynamical model of the motion behavior of a Pioneer I robot. Models can be learned similarly for other robots with the respective data at hand.

The Pioneer I Robot

The dynamic state of a Pioneer I robot at a certain time t is given by the quintuple

$$\zeta(t) = \langle x, y, \varphi, V_{tr}, V_{rot} \rangle \tag{3.9}$$

where x and y are coordinates in a global system (in meters), φ is the orientation of the robot (in degrees, $\in [0, 360)$) and V_{tr} (V_{rot}) is the translational (rotational) velocity in meters per second (degrees per second $[-180, 180]$). The robot control system issues driving commands

$$a = \langle V_{tr}, V_{rot} \rangle \tag{3.10}$$

at a frequency of 10 Hz. The parameters of the driving command denote the desired target velocity. Naturally, the robot needs some time to reach the target velocity and it will not be able to drive exactly at the desired velocity.

Before we start to learn a simulator we take a look at the dynamical behavior of the robot as measured by its odometry sensors. Figure 3.5(a) depicts the rotational velocity V_{rot} of a Pioneer I robot that starts with $V_{rot} = 0$ and is given the control command $a = (0, 45)$ repeatedly every $\frac{1}{10}s$. As we can see, the actual rotational velocity (after a period of acceleration) is around 15 degrees per second only with a high degree of oscillation. Figure 3.5(b) shows a similar behavior for the control command $a = (0, 90)$: The actual rotational velocity is around

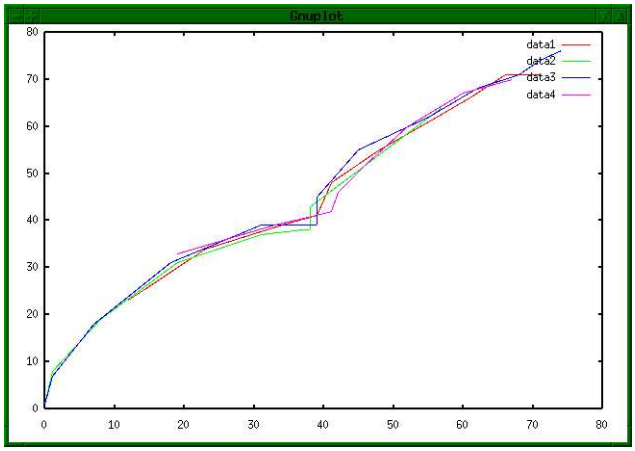


Figure 3.6: The resulting translational velocity (cm/s) depending on the robot's previous velocity (frequency 10 Hz). Four runs of a Pioneer I robot are shown. In each experiment the robot starts at $V_{tr} = 0$ and is constantly given the control command $a = (0.75, 0)$.

30 degrees per second. In figure 3.6 we can see how the translational velocity V_{tr} of a Pioneer I robot changes over time if the robot starts with $V_{tr} = 0$ and is given control commands $a = (0.75, 0)$ repeatedly. Four experiments were done under the same conditions[1].

The conclusions we can draw from the above measurements are

(1) Either the odometry sensors of the robot are extremely noisy or the hardware-unit responsible for the control of the velocity is oscillating enormously (see figures 3.5). Since it is unlikely that the velocity changes that quickly we assume the sensors are noisy.

(2) The resulting rotational velocity is around one third of the desired rotational velocity (see figures 3.5). The relation may even be nonlinear.

(3) The dynamical behavior of the robot is inherently indeterministic: The four experiments depicted in figure 3.6 lead to different results under the same conditions.

(4) The acceleration of the velocity is nonlinear and cannot be described by a trivial hand-coded function (see figure 3.6).

The consequences these factors have on the development of our simulator are

(1) It would be inappropriate to map the oscillating behavior into our simulation. We are interested in a less volatile function that approximates the curves of the figures 3.5(a) and 3.5(b). This will make our simulation smooth *and* realistic. We can learn such an approximation if we simply use all training data (including oscillation) and perform neural learning (see appendix A.2). While the error on the training data may remain high even after extensive training the trained function will be an approximation appropriate for simulation, especially if we use a sufficient amount of training data and validation data.

(2) The fact that the robot actually only reaches around one third of its desired rotational velocity affects us not. We can learn this behavior.

(3) The indeterministic behavior can be treated the same way the oscillation is treated. If wanted we can add noise to the simulation or explicitly learn the noise to achieve an indeterministic behavior in simulation too.

(4) The fact that acceleration, in practice, could be any nonlinear function forces us to use learning methods that can deal with nonlinear relations. We will use neural networks (see appendix A.2 for details) because of their advantageous run-times in application.

[1] at least from a human point of view the conditions were the same

Dead Time Delays All robots are dealing more or less with a dead time. This means that at the moment a robot is performing a low level control command it has already sent control commands (e.g. translational and rotational target velocity) for further movements. In the case of Pioneer I robots the dead time is around $300ms$ (see figure 5.12(a), p. 71) while the controller accepts ten commands per second. In simulation the dead time must be considered as well.

Learning the Simulation

The dynamical model for the change in state from the current state $\zeta(t)$ to the successor state $\zeta(t + \Delta t_\zeta)$ used by the simulator is acquired by learning the mapping of equation (3.7). In case of a Pioneer I robot where the cycle times are set to $\Delta t_\zeta = 100ms$ and $\Delta t_a = 100ms$, and the dead time delay is $300ms$ (= three cycles) this mapping will be specified by

$$\mathcal{P} = \begin{cases} \mathcal{S} \times \mathcal{A} \to \mathcal{S} \\ \zeta(t), a(t - 3 \cdot \Delta t_a) \mapsto \zeta(t + \Delta t_\zeta) \end{cases} \tag{3.11}$$

$\mathcal{P}$ is learned from experience, that is recorded data from real robot runs. But before we perform neural learning we preprocess the learning data. The input space of $\mathcal{P}$ is 7-dimensional (5-dim. state plus 2-dim. action). Considering the current state $\zeta(t)$ at $x = 0, y = 0$, and $\varphi = 0$ in a local system we can reduce the input dimension to 4 by converting the successor state's ($\zeta(t + \Delta t_\zeta)$'s) x, y, φ into this local system (this means we regard $\Delta x, \Delta y, \Delta\varphi$ instead of x, y, φ). So we simplify $\mathcal{P}$ to

$$\mathcal{P} : \langle V_{tr}(t), V_{rot}(t), V_{tr}, V_{rot} \rangle \mapsto \langle \Delta x, \Delta y, \Delta\varphi, V_{tr}(t + \Delta t_\zeta), V_{rot}(t + \Delta t_\zeta) \rangle \tag{3.12}$$

Since $\Delta x, \Delta y$, an $\Delta\varphi$ can be approximated from $V_{tr}(t), V_{rot}(t)$ and $V_{tr}(t + \Delta t_\zeta), V_{rot}(t + \Delta t_\zeta)$ we will further simplify $\mathcal{P}$ to

$$\mathcal{P} : \langle V_{tr}(t), V_{rot}(t), V_{tr}, V_{rot} \rangle \mapsto \langle V_{tr}(t + \Delta t_\zeta), V_{rot}(t + \Delta t_\zeta) \rangle \tag{3.13}$$

Our simulator learns this mapping using standard multi layer neural networks [Hecht-Nielsen 1990, Hertz *et al.* 1991, Bishop 1995, Rojas 1996] (see appendix A.2) with 4-10-16-10-2-topology, supervised learning with the RPROP algorithm [Riedmiller and Braun 1993] and early stopping [Sarle 1995]. During data acquisition we have executed a wide variety of navigation scenarios covering even abrupt changes in velocity and orientation to comply with the requirements of high-speed navigation. We have collected a total of more than 100,000 training patterns from runs with a real Pioneer I robot. 80% of this data has been used

for training while 20% are used as validation data. Although the learning time amounted to a few hours on a 800 MHz machine the computational amount of the neural network while running the simulation is infinitesimal.

Considering the Dead Time Delay In consideration of the dead time delay of $300ms$ in simulation we write low level control commands in a queue and wait $300ms$ before executing the commands.

Accuracy We evaluated on about 100 trajectories, each driven with a real Pioneer I robot ten seconds in length. None of them were used for learning. We found an average error of 2% in the simulation of position and orientation. This error largely results from the inherent indeterministic behavior of the robot controller (see figure 3.6).

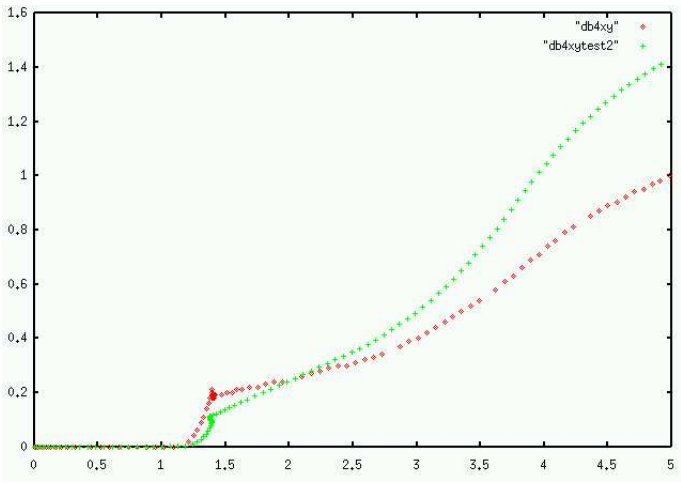


Figure 3.7: A navigation including extreme changes in translational and rotational velocity. The picture shows the simulated and the real path of the robot (coordinates of position in meters).

This indeterministic behavior itself usually accounts for an average error of around one percent. After extreme navigation situations where the translational velocity was set from full to zero and the rotational velocity was set from zero to full (or both the other way around) sometimes an error of up to 10% in orientation occurred (see figure 3.7 for example). These cases are likely the hardest to predict at all and not very common in real robot navigation. Figure 3.8 shows real robot data and predicted data of the simulator in two prevalent trajectories. Obviously the match of predicted and real robot data looks satisfying. The remaining small errors in simulation are balanced by the high-level control loop running at a frequency of 10 Hz in real-time.

Scalability of the Method In addition to our Pioneer robots the simulation method introduced above can be used for any other real robots such as B21 or Nomad robots.

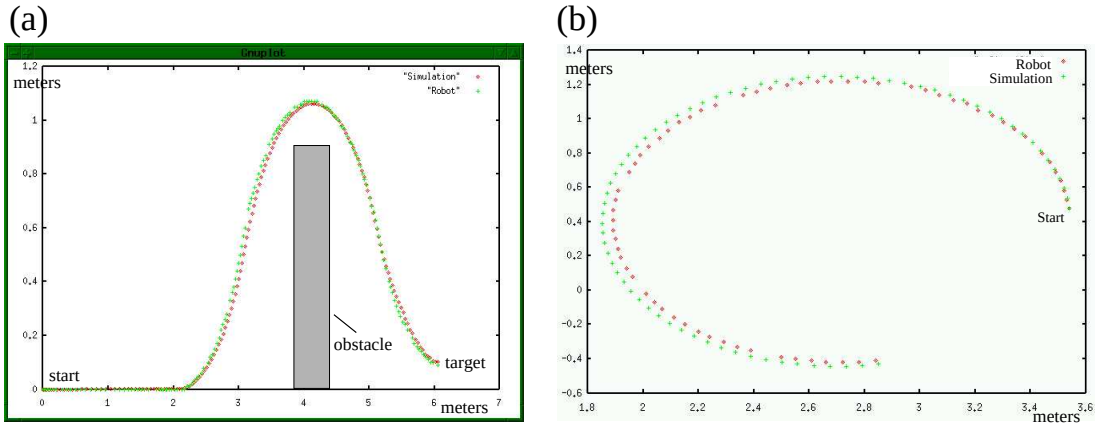


Figure 3.8: A comparison between the position data of a real robot and the data simulated. Subfigure (a): A robot driving a jink around an obstacle. Subfigure (b): A robot driving a curve.

Comparison with the RoboCup Simulation League

The main difference between our simulation and the well known RoboCup soccerserver simulation environment [Kitano *et al.* 1997, Noda *et al.* 1998] is that we can simulate a team of real Pioneer I robots while soccerserver simulates a hybrid, in some aspects human-like, soccer player. Our robot soccer team (the *AGILO RoboCuppers* [Beetz *et al.* 2002]) consists of four Pioneer I robots [Pioneer 1998] that have a number of serious disadvantages concerning their dynamical behavior compared to an agent of the soccerserver:

- Pioneer robots (like a number of other real robots) cannot hold the ball if it comes not directly into their ball-guiding device.
- Path planning with real robots becomes more complicated than in the soccerserver where the field is about 105 times 70 meters and a player has a diameter of 0.3 meters only.
- A Pioneer robot (like any other real robot) has to deal with time delays, nonlinear acceleration, and noisy sensors.

Moreover if the ball is fast enough it can go through a player in soccerserver. It can be given a velocity vector by a player having it in his kicking range. Teams like the *Karlsruhe Brainstormers* [Riedmiller and Merke 2002] have shown that in the soccerserver environment learning algorithms perform beautifully and, in fact, a lot better than a human being can control a player with a joystick.

Chapter 4

Experience-Based Control

4.1 Introduction

Many machine control skills are difficult and tedious to code by hand. Preferably, such skills should be learned automatically by the machine – for instance, through experience-based learning techniques. Unfortunately, many of these learning tasks in the context of skill acquisition are very difficult to solve: Their state spaces are high dimensional, variables and command parameters are continuous values, there may be a lack of training data, the exploration may be unsuccessful, and the influence of noise may be high (see section 2.2). This situation calls for pragmatic solutions to machine control problems.

Properties of Experience-Based Control In machine control we are looking for a mapping

$$\pi : \begin{cases} \mathcal{S} \rightarrow \mathcal{A} \\ \zeta(t) \mapsto a(t) \end{cases} \tag{4.1}$$

called *policy* that maps a current state, $\zeta(t)$ to the action to be performed, $a(t)$. We want to know how the machine should behave, which action should be performed in which situation to reach the target state (as soon as possible). In the following we will describe the main properties of our approach of *experience-based control* (EBC) and explain why they are necessary for the successful learning of machine control tasks.

- **Learning from Experience and Interaction** To obtain control software that copes with the problems of real world applications we need to base our learning on experience. Learning depends on feedback from trials performed

in the real world. Software developed without any feedback from interaction with the environment has little chance to work reliably.

- **Inductive Learning** Since we want our control software to work not only in the cases observed during the acquisition of experience but also in cases not observed before, we have to apply inductive learning to obtain a high degree of generalization.

- **Automatic, Indirect Supervised Learning** Supervised learning means to use a teacher who has to provide training patterns for the learning algorithm in order to improve its performance. Since it is (1) very laborious and (2) in some cases impossible to use a human teacher to generate training patterns we want the patterns to be generated automatically.

- **Exploration with an Initial Policy** To automatically generate training patterns we explore the state space by performing actions that are chosen (1) randomly and (2) based on initial knowledge. This will enable us to exploit a priori knowledge and to explore the state space randomly too. Before performing an action the respective successor state is unknown. The obtained trajectory of states is used for learning.

- **Non-Incremental Learning** During exploration we receive no immediate feedback through rewards. This avoids (1) the incremental summation of errors, (2) the mono-directed exploration of the state space based on previous experiences, and (3) the forgetting of experiences acquired long ago. Learning is completely performed offline. In applications with dynamic environments the dynamic components must be included in the state space because they cannot be learned online.

In experience-based control we try to learn from trajectories of past explorations by generalization. Our methods rely on a three step process: First, states visited during *exploration* are assigned values dependent on their temporal distance to the target state. In the second step, these values, together with the respective states, are used for *learning* a temporal distance metric for the state space. Function approximators support the learning process. Finally, we *exploit* the learned metric in order to control the machine. Thus exploration, learning, and exploitation are strictly divided. For some applications we can even learn a direct mapping from state space to action space instead of a distance metric. The different techniques for exploration and learning will be introduced in the sections 4.3 and 4.4, respectively. But first, we describe the pre-exploration as a necessary precondition for all following methods.

4.2 Pre-Exploration

As a precondition for applying the learning methods that will be introduced below we have to learn projection-functions that enable us to efficiently explore the state space and find the successor (or predecessor) state of any state-action-pair.

Forward-Projection The function used for forward-projection is given by

$$\mathcal{P}^+ : \begin{cases} \mathcal{S} \times \mathcal{A} \to \mathcal{S} \\ \langle \zeta(t), a(t) \rangle \mapsto \zeta(t+1) \end{cases} \tag{4.2}$$

where a state, $\zeta(t)$ and an action at time t, $a(t)$ are mapped to a successor state, $\zeta(t+1)$. The forward-projection is used for predicting the successor state of a proposed action. The proposed action can then be evaluated according to the appropriateness of its successor state. Further $\mathcal{P}^+$ can be used for means of simulation if $\bar{\mathcal{A}}$ contains one action only (see chapter 3). This is very important if the costs (in terms of time and/or money) for exploration with the real machine are high. In this context, we assume $\mathcal{P}^+$ to be a *deterministic* function.

Backward-Projection Similar to the forward-projection (equation (4.2)) we define a function that implements a backward-projection:

$$\mathcal{P}^- : \begin{cases} \mathcal{S} \times \mathcal{A} \to \mathcal{S} \\ \langle \zeta(t+1), a(t) \rangle \mapsto \zeta(t) \end{cases} \tag{4.3}$$

Here, a state, $\zeta(t+1)$ and its preceding action at time t, $a(t)$ are mapped to the predecessor state, $\zeta(t)$. Like $\mathcal{P}^+$, we assume $\mathcal{P}^-$ to be a deterministic function. Fortunately, for the majority of real world control processes, $\mathcal{P}^-$ can be regarded as a deterministic function. At least $\mathcal{P}^-$ is unimodal distributed with a small variance. The backward exploration following in section 4.3.2 is based on this backward-projection.

Learning the Projection-Functions Accurate projection-functions are essential for the reliability of the control methods described in this chapter. They work as a basis for the learning of control behaviors. The requirements for training data as explicated in section 2.2.1 apply to the training of the projection-functions in particular. To obtain functions that implement nearly the same behavior as a real machine we need large amounts of training data covering the entire state space.

Either training patterns are acquired by simply performing actions on the real machine and recording all actions and states over time or training patterns are

acquired by doing the same but in simulation. While the first approach will likely lead to more precise projection-functions the second one allows for a more efficient collection of training data. Patterns contain two consecutive states, $\zeta(t)$ and $\zeta(t+1)$, and the action responsible for the respective change in state, $a(t)$:

$$\begin{array}{ll} \text{pattern for } \mathcal{P}^+ & \text{pattern for } \mathcal{P}^- \\ \langle \zeta(t), a(t) \rangle, \zeta(t+1) & \langle \zeta(t+1), a(t) \rangle, \zeta(t) \end{array} \tag{4.4}$$

The data at hand is learned by means of function approximation. Useful approximators for learning projection-functions include neural networks (see appendix A.2) and networks of radial basis functions (see appendix A.3).

4.3 Exploration

The exploration of the state space of a machine provides an opportunity to gather information (1) about the states the machine can reach in extreme situations and, even more important, (2) about the behavior of the machine depending on the actions that were executed before. Since we want to perform experience-based learning, for us, the exploration of the state space is the key to acquiring experience. In the following subsections we first describe the standard way of exploration (forward exploration) and then introduce sophisticated exploration techniques to overcome problems occurring in forward exploration.

4.3.1 Forward Exploration

The most plausible way of exploration is the simple forward exploration. The machine starts at a preselected or randomly defined start state ζ_{start} and performs actions according to a well defined initial policy. All states visited during this exploration are recorded. As soon as the machine reaches a state fulfilling the constraints of the target state the exploration is ended successfully (figure 4.1(a) depicts such a scenario). Then the data of the recorded states is used to generate learning data to train a functioning controller. To perform a forward exploration in simulation the forward-projection $\mathcal{P}^+$ (equation (4.2)) is inherently needed.

Initial Policies All explorations that reach the target are called successful. The success of an exploration, however, depends on the initial policy employed for the selection of actions. This policy can be to choose any action of the action space by random. Alternatively, it can be hand-coded. This is also the easiest way to include a priori knowledge. It is advisable to use a hybrid policy which chooses partly by random and which has a component that sometimes chooses

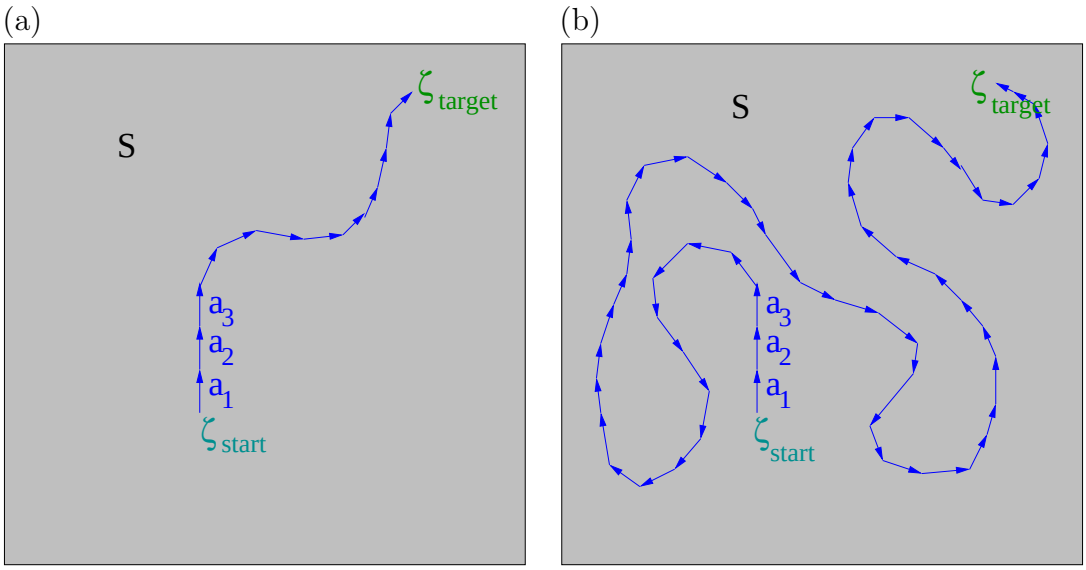


Figure 4.1: Subfigure (a): A forward exploration in continuous state space. ζ_{target} is reached after a number of actions $(a_1, a_2, ...)$. Subfigure (b): After a long forward exploration ζ_{target} is reached. There probably would have been a better way.

greedily based on an a priori defined simple distance metric that is assumed to be not entirely wrong.

Problems

Detours Let us assume that some trajectories from exploration runs are given. These trajectories lead from different states in the state space to the target state. Since we usually do not know a reasonable distance metric for the state space we might not be competent to tell if the performed successful exploration runs were detours or not. Even if we think they are detours we cannot directly extract information about more efficient trajectories to the target. Since we want to create an efficient controller we have to take care that we do not use too many inefficient trajectories for learning. Figure 4.1(b) shows a detour assuming the Euclidean distance metric.

Unsuccessful Exploration Naturally, not all exploration runs will reach the target space (see figure 4.2(a) for an example), especially if the target space is very small compared to the whole state space or even consists of one single target state only. In fact, usually there are many more exploration runs that do not reach the target state than runs that do. But if none of the exploration runs reaches the target state we will not be able to learn how to control the machine

in order to reach the target state from this experience. We essentially must obtain trajectories from exploration that lead to the target state.

Insufficient Exploration of Interesting Regions Furthermore there is the question if the state space was covered sufficiently during exploration. Maybe there are regions which are explored but need to be explored more in detail because changes in state are enormous even for marginal changes in the actions.

Solutions

Obviously, the more exploration runs we do the smaller the problems will be: (1) The more runs we do, the greater the chance of obtaining successful explorations. (2) Relying on a great number of exploration runs the error of single detours will be balanced. However, we do not have unlimited training data and time.

Restrictions on Trajectories To avoid detours the following restrictions for explorations are proposed:

(1) The time per exploration, and with it the length of the corresponding trajectory, should be limited. The length of the trajectory can be increased iteratively if it is not sufficient. To find the optimal length requires some domain specific knowledge but is manageable.

(2) Actions should change smoothly. In the context of choosing a direction this means not to go north directly after going south but to go, for example, north-west in the next step. In particular two consecutive actions should not be *inverse*. This means that performing two actions should not keep the state almost constant. Further, a smooth change of actions will lead to a smooth behavior of the learned controller. The user can easily define criteria in order to determine if an action is inverse or not.

(3) Regions already explored during the current exploration run should be widely avoided.

However, to explore regions of interest more in detail we have to explore these regions more than one time (as proposed by [Moore and Atkeson 1995] for reinforcement learning in discrete state spaces).

Other Exploration Techniques While all the above are modifications of the forward exploration the following techniques are based on different ideas.

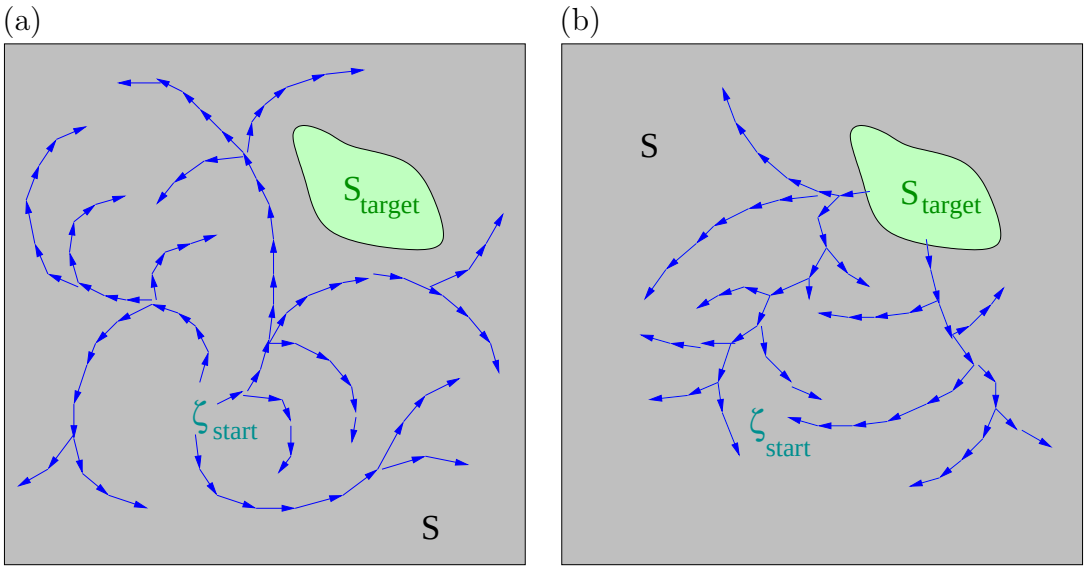


Figure 4.2: Subfigure (a): From the start state ζ_{start} no successful exploration can be found that leads to the target space $\mathcal{S}_{target}$. Subfigure (b): The backward exploration provides trajectories leading from some states in state space to the target space $\mathcal{S}_{target}$.

4.3.2 Backward Exploration

To overcome the problem of unsuccessful exploration we propose the method of backward exploration. After unsuccessful exploration runs we cannot give any information about the distance to the target state. Backward exploration works the same way as forward exploration but the exploration starts at the target state. The state space is then explored backwards from the target state by using the backward-projection $\mathcal{P}^-$ (equation (4.3), figure 4.2(b)). Inherently this can only be done in simulation but not in the real world. The goal of backward exploration is to cover the state space with trajectories from explorations starting at the target state. It is not necessary to reach any certain start state because the temporal distance to the target is given for a number of states and can be computed by generalization of these states for any other state.

Since in backward exploration *all* exploration runs are successful (because they all connect non-target states with the target state) we need less training time for the same number of successful explorations. This leads, dependent on the application at hand, to an enormous speed-up in learning. In fact, in cases where no successful explorations can be made using forward exploration, it is essential to use backward exploration. However, there might be very few applications where the backward-projection $\mathcal{P}^-$ is not deterministic and therefore difficult to use.

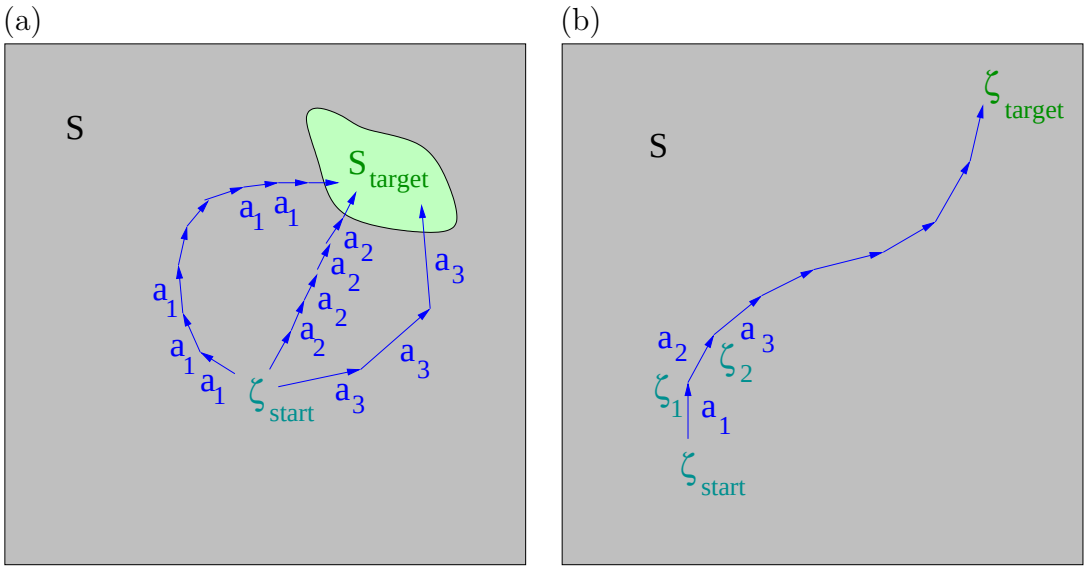


Figure 4.3: Subfigure (a): Fixed-action exploration. The action chosen is kept fixed during the entire exploration. Subfigure (b): Supervised exploration. A predefined sequence of actions $(a_1, a_2, ...)$ is performed from ζ_{start}. The resulting state at the end of the exploration is called target state.

4.3.3 Fixed-Action Exploration

In applications where trajectories that lead to the target are known in advance and we want to know how to get there as quickly as possible we can use a different type of exploration style. Imagine, for instance, in the real world there is a mountain to climb and there are a number of hiking trails that lead to the mountain's peak. We may want to know which hiking trail to use to reach the target as quickly as possible. In this case the action space is discrete and consists of a number of trails. The action is *fixed for the whole exploration run*. The state space contains features that describe the characteristics of the current situation. In the example this could be the length of the different trails, the altitude (maybe a particular trail requires us to walk down and up again), and so on. In terms of machine control we could have the choice between different control policies that take different amounts of time to reach a target state. In fixed-action exploration we assume a *finite* number of possible actions that lead to the target and we perform them *all* in order to find the most efficient one.

4.3.4 Supervised Exploration

Sometimes we want to teach a particular behavior to a controller by providing a sequence of actions without knowing the explicit target state in advance. But without an explicit target state all the above exploration techniques will not work.

The idea behind supervised exploration is to perform a sequence of actions until we reach a state that we think that is useful to know how to reach. Then we name the reached state *target state* and compute all previous states relative to this target state. This means we effectively redefine the origin of the state space. All states visited during the exploration run are regarded from the target state's point of view.

Unfortunately this method only works if the state space is invariant in all dimensions. This means that the change in state $\Delta\zeta(\zeta(t), a(t), \Delta t)$ depends on the action, $a(t)$ and a constant time value, Δt only and is independent of the current state, $\zeta(t)$. Otherwise we cannot redefine the origin of the state space.

4.4 Learning Policies

After having introduced different methods for the exploration of the state space we now take a look at how to learn a policy-function (equation (4.1)). Depending on a priori knowledge and complexity of the task at hand different approaches will be used to construct a policy function. First, we will shortly distinguish between local and global optimization (subsections 4.4.1 and 4.4.2) before we describe the single approaches in detail.

A value-function (subsection 4.4.3) receives a state as an input only. Additionally, we need a projection-function to implement the policy-function. The approximation of a Q-function (subsection 4.4.5) that receives a state and an action as an input is more difficult to be implemented but does not require a projection-function. Finally, the direct approximation of the policy-function (subsection 4.4.6) is most straightforward but in many cases not possible because of its complexity.

4.4.1 Local Optimization

A number of control problems can be solved by using only local optimization: Moving to a target place inside a room without any obstacles is such a case: We can act greedily and take an action (direction) that minimizes the distance to the target state. The action that is optimal from a local point of view is the optimal action from a global point of view, too. However, we should not use local optimization if there are obstacles in the room. In this case we might walk into a direction that is optimal from a local point of view but not from a global point of view because we might come across some obstacles later on by following this direction.

If a distance metric that gives the distance to the target for any state is available (for instance the Minkowski metric), the target state can be easily reached, pro-

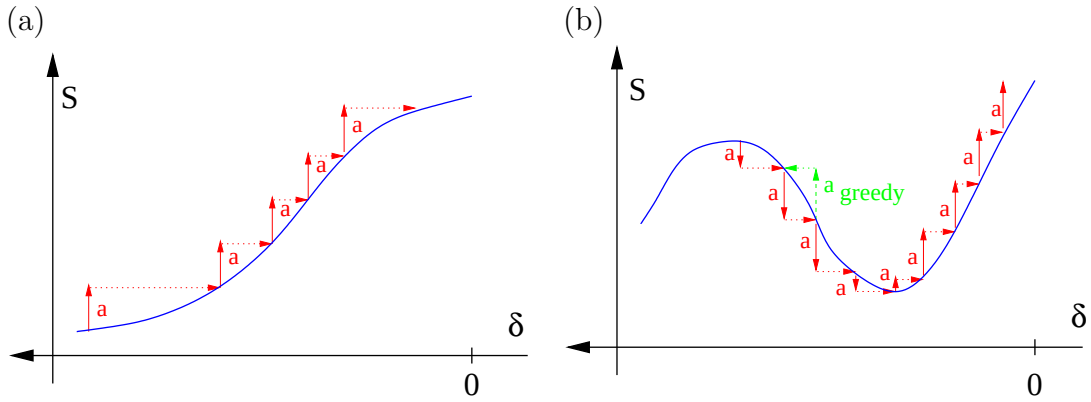


Figure 4.4: Subfigure (a): The way to the target state is strictly monotone. Subfigure (b): The way to the target state is not strictly monotone regarding the depicted dimension of the state space. Performing greedy actions (arrow marked greedy) might not lead to the target state.

vided that the distance metric δ is strictly monotonic decreasing. If this is the case, we simply always choose the action that leads to a state with the minimal distance to the target:

$$\pi(\zeta(t)) = \arg\min_{a \in \mathcal{A}} \delta(\mathcal{P}^+(\zeta(t), a), \zeta_{target}) \tag{4.5}$$

In case of the Minkowski metric local optimization works well if the optimal way to the target state is strictly monotonic increasing (or decreasing) in all dimensions of the state space.

4.4.2 Global Optimization

For a great number of control problems we do not know a distance metric that fulfills the requirements mentioned above. Here, we cannot use equation (4.5) with a simple metric (for instance the Minkowski metric). The way to the target state does not have to be strictly monotonic increasing (or decreasing) in all dimensions of the state space. Figure 4.4(a) shows a state space where the way to the target state is strictly monotone. In Figure 4.4(b) the way is not strictly monotone. Therefore simple metrics should not be used for finding a way. In the following we introduce different methods for learning the policy. While the approximation of the value-function and the approximation of the Q-function aim at an indirect learning of the policy-function it can be learned directly by an approximation of the policy-function itself.

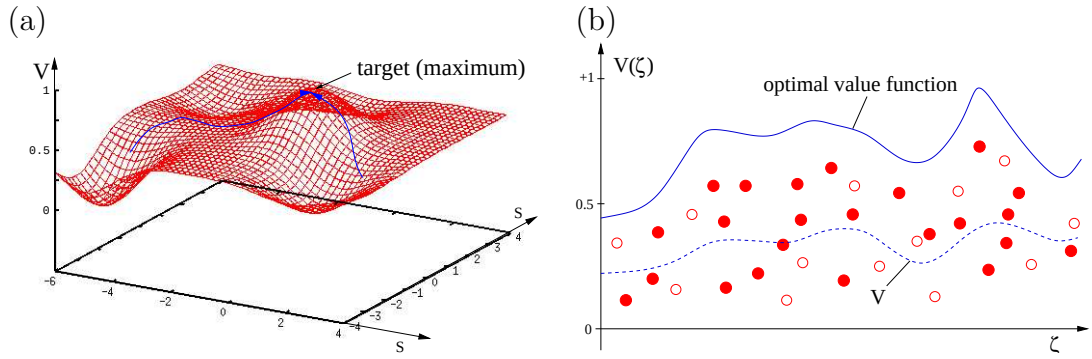


Figure 4.5: Subfigure (a): The maximum of the value-function corresponds to the target state. Minima denote undesirable states. Subfigure (b): The generalized function $\mathcal{V}$ is a lower bound for the optimal value-function and has a shape similar to it.

4.4.3 Value-Function Approximation

As mentioned in section 4.1, we want to learn a temporal distance metric in the state space $\mathcal{S}$. According to the syntax of a value-function in reinforcement learning[1] (see section 7.1) we call this metric $\mathcal{V}$ and define that high values correspond to states close to the target and low values correspond to states far away. To have an upper bound for the time to reach the target state, ζ_{target} or the target space, $\mathcal{S}_{target}$ from a particular start state we have to navigate there through state space and record the time needed. An upper bound for the time corresponds to a lower bound for the optimal value function. Once we know the value for all states by having learned a continuous value-function we perform gradient ascent to get to the target state (see figure 4.5(a)).

Exploration For the approximation of the value-function we use forward exploration (see section 4.3.1) and backward exploration (see section 4.3.2, figure 4.6(a)). As we explore states $\zeta(t)$ at time t until we arrive at the target space $\mathcal{S}_{target}$ we can assign values to these states:

$$\mathcal{V}(\zeta(t)) = \begin{cases} +1 & \textit{if } \zeta(t) \in \mathcal{S}_{target} \\ \gamma \cdot \mathcal{V}(\zeta(t+1)) & \textit{else} \end{cases} \tag{4.6}$$

If we reach $\mathcal{S}_{target}$ at time t we assign $\mathcal{V}(\zeta(t)) = 1$ to $\zeta(t)$ and $\mathcal{V}(\zeta(t-i)) = \gamma^i$ to all predecessor states where γ is a discount factor ($\gamma \in (0,1)$, see figure 4.6(b)). Thus we get patterns

$$\zeta(t), \mathcal{V}(\zeta(t)) \qquad \zeta(t) \in \mathcal{S}, \;\; \mathcal{V}(\zeta(t)) \in (0,1] \tag{4.7}$$

[1] In reinforcement learning $\mathcal{V}$ is defined as the sum of reinforcements received when following some fixed policy to a terminal state.

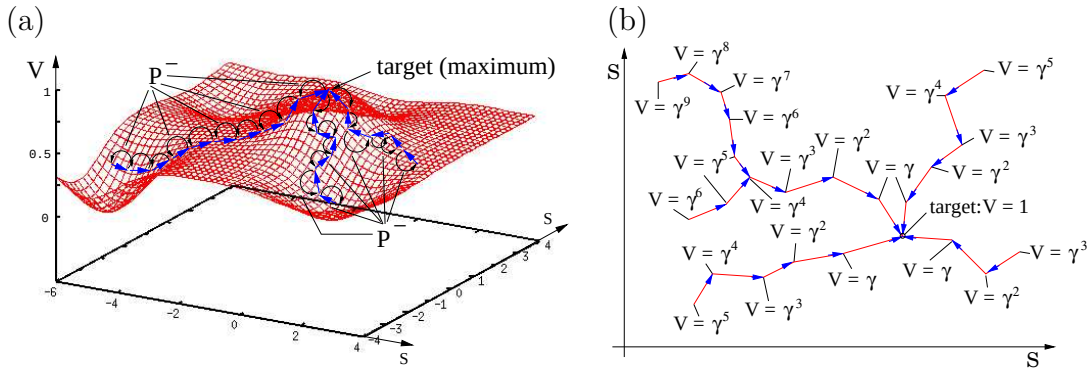


Figure 4.6: Subfigure (a): Using the backward-projection the state space is explored starting at the target state. Subfigure (b): During exploration the value of each state visited is computed according to equation (4.6).

that we can use for learning a lower bound for the optimal value-function. The idea is that having a sufficient number of those patterns (unlimited in theory but a reasonable number in practice) we will be able to approximate a function $\mathcal{V}$ whose local extrema are located at almost the same points in state space as those of the optimal value-function (see figure 4.5(b)). If we randomly explore the state space a point near the target space will more likely get a high value than a point far away.

Learning The value-function $\mathcal{V}$ is represented by a function approximator f_{approx} that maps a state $\zeta(i)$ to a real value:

$$\mathcal{V} : \begin{cases} \mathcal{S} \rightarrow I\!R \\ \zeta(i) \mapsto f_{approx}(\zeta(i)) \end{cases} \tag{4.8}$$

The approximator f_{approx} is learned using the patterns of equation (4.7). We can use different function approximators (see appendix A for function approximators). [Bertsekas and Tsitsiklis 1996] propose to use multi layer artificial neural networks [Hecht-Nielsen 1990, Hertz *et al.* 1991, Bishop 1995, Rojas 1996] for the approximation of a value-function (in reinforcement learning). We mainly prefer multi layer neural networks (see appendix A.2) and networks of radial basis functions [Powell 1985, Broomhead and Lowe 1988, Poggio and Giorosi 1989, Moody and Darken 1989] (see appendix A.3). RBF-networks are appropriate because we can easily build attractors around trajectories using them. Both methods, neural networks and RBF-networks, can deal with continuous input and output spaces, they can approximate highly complex functions, and they are very efficient during exploitation.

Using a sufficient number of training and validation patterns for learning we are able to detect if a low measured value corresponds to a local minimum in state

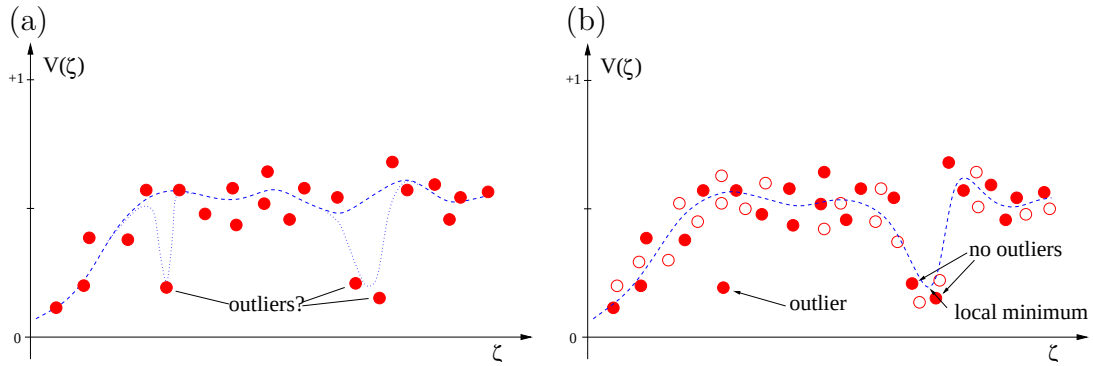


Figure 4.7: Detecting outliers in state space: Subfigure (a): Which of the measured values are outliers? Subfigure (b): The dashed line is learned from training data (filled circles) using validation data (other circles). Outliers are identified.

space or if the exploration was a detour (see figure 4.7). The computational amount for learning is around $O({n_{expl}}^d)$ where n_{expl} is the estimated amount for the sufficient exploration per dimension and d is the number of dimensions of the state space. However, as the size of the state space grows exponentially with the number of dimensions the amount needed for learning in that space does as well.

Exploitation Assuming we have managed to learn the value-function $\mathcal{V}$ we need to know which action a in a given state $\zeta(t)$ leads to the best successor state $\zeta(t+1)$. This is done in two steps: First, we heuristically generate a number of possible actions. In case of a continuous action space this can be done by an equidistant covering of the action space. An interpolation between different actions is possible too. In case of a finite and relatively small discrete action space we can evaluate all possible actions. Secondly, we predict the supposed successor state for each possible action a and choose the action corresponding to the best-valued successor state (see figure 4.8). Thus our policy π is defined

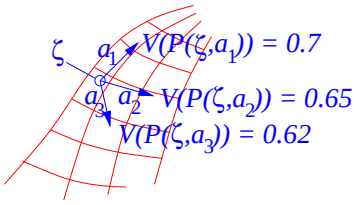


Figure 4.8: Exploitation: We choose the action promising the highest value for the successor state (in this case a_1).

$$\pi(\zeta(t)) = \arg\max_{a \in \mathcal{A}} \mathcal{V}(\zeta(t+1)) = \arg\max_{a \in \mathcal{A}} \mathcal{V}(\mathcal{P}^+(\zeta(t), a)) \tag{4.9}$$

Here, $\mathcal{P}^+$ is the forward-projection defined in section 4.2 (equation (4.2)).

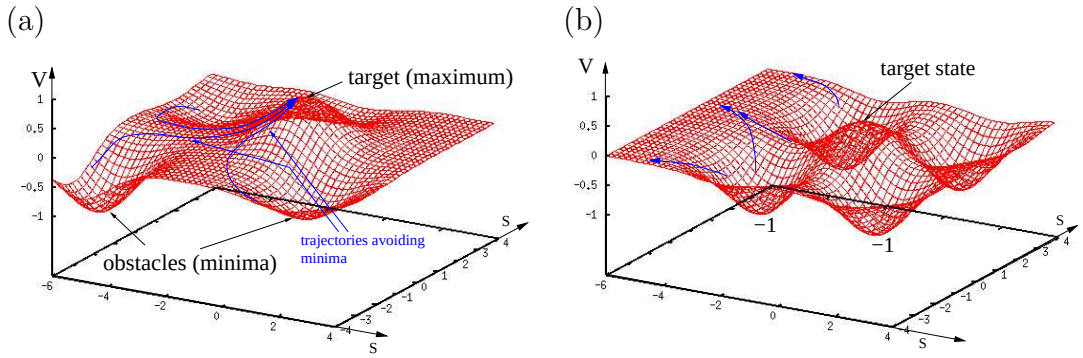


Figure 4.9: Subfigure (a): Using two value-functions $\mathcal{V}^+$ and $\mathcal{V}^-$. Undesirable states (obstacles) are minima. They are circumnavigated. Subfigure (b): The target state is surrounded by states with minimal value. It may not be possible to find a way to the target state by using the policy of equation (4.13).

4.4.4 Using Two Value-Functions

So far, we cannot specify undesirable states (states we do not want to visit) with the value-function we introduced in the last section.

Exploration Equally to assigning positive values to states of successful explorations (explorations that lead to the target state in the end) we can assign negative values $\mathcal{V}^-$ to states of explorations that lead to undesirable states. In the latter case we assign values according to the following equation:

$$\mathcal{V}^-(\zeta(t)) = \begin{cases} -1 & if\ \zeta(t) \in \mathcal{S}_{undesirable} \\ \gamma \cdot \mathcal{V}^-(\zeta(t+1)) & else \end{cases} \tag{4.10}$$

where $\mathcal{S}_{undesirable}$ is the space containing all undesirable states and γ is a discount factor ($\gamma \in (0,1)$). If we reach an undesirable state we penalize $\zeta(t)$ with the value $\mathcal{V}(\zeta(t)) = -1$ and all its predecessor states according to γ. The patterns obtained contain

$$\zeta(t), \mathcal{V}^-(\zeta(t)) \qquad \zeta(t) \in \mathcal{S},\ \mathcal{V}^-(\zeta(t)) \in [-1, 0) \tag{4.11}$$

Learning The above patterns are used to learn the function

$$\mathcal{V}^- : \begin{cases} \mathcal{S} \to I\!R \\ \zeta(t) \mapsto f_{approx}(\zeta(t)) \end{cases} \tag{4.12}$$

by means of function approximation (the same approximators as for $\mathcal{V}^+$ can be used). In addition to $\mathcal{V}^-$ we have to learn $\mathcal{V}$ (equation (4.8)) too. In the following we name $\mathcal{V}^+ = V$.

Exploitation When we have learned the value-functions $\mathcal{V}^+$ and $\mathcal{V}^-$ (for target states and undesirable states respectively) we determine which action a leads to the best successor state $\zeta(t+1)$ for a given state $\zeta(t)$. This can be done by using the policy

$$\pi(\zeta(t)) = \arg\max_{a \in \mathcal{A}} [\mathcal{V}^+(\mathcal{P}^+(\zeta(t), a)) + \mathcal{V}^-(\mathcal{P}^+(\zeta(t), a))] \tag{4.13}$$

Then, the function $[\mathcal{V}^+ + \mathcal{V}^-]$ is used to find the action corresponding to the best-valued successor state. This avoids undesirable states (see figure 4.9(a)).

But what happens for example if we define a number of undesirable states (with minimal value -1) around the target state? This could correspond to a robot which is to navigate through a narrow passage. The resulting value-function may have local maxima and the robot may not find a way to its target avoiding the states with negative values (see figure 4.9(b)). This problem is similar to problems occurring in path planning using potential fields. In addition to numerous variations of the potential field method that overcome the problem of local maxima (see [Barraquand *et al.* 1992] for example) we propose a special solution to this problem in our context. We use a threshold $\Theta_{\mathcal{V}^-}$ that denotes the absolute minimum of $\mathcal{V}^-$ we are willing to tolerate ($\Theta_{\mathcal{V}^-} \in [-1, 0]$). Thus $\Theta_{\mathcal{V}^-}$ is a well understood parameter. The policy is to use $\mathcal{V}^+$ as defined in section 4.4.3 and to select an action only if the value of $\mathcal{V}^-$ for its successor state is better than $\Theta_{\mathcal{V}^-}$:

$$\pi(\zeta(t)) = \arg\max_{a \in \mathcal{A}} \mathcal{V}^+(\mathcal{P}^+(\zeta(t), a)) \cdot Eval[\mathcal{V}^-(\mathcal{P}^+(\zeta(t), a)) > \Theta_{\mathcal{V}^-}] \tag{4.14}$$

Herein $Eval[\cdot > \cdot]$ is an evaluation function that tells us if one number is bigger than another one:

$$Eval[b_1 > b_2] = \begin{cases} 1 & \textit{if } b_1 > b_2 \\ 0 & \textit{else} \end{cases} \tag{4.15}$$

This approach guides us not only towards the target space. It even allows us to define undesirable states (or spaces) we definitely do not want to visit.

4.4.5 Q-Function Approximation

If we do not want to use the forward-projection $\mathcal{P}^+$ (or if we *cannot* use it because we cannot compute it for some reason) we have to use another approach. For a

number of applications it is possible to learn a Q-function[2] that maps from state *and* action to a value that we call Q-value then. If we are able to learn such a function we can use it similar to using a value-function:

$$Q(\zeta(t), a) = \mathcal{V}(\zeta(t+1)) = \mathcal{V}(\mathcal{P}^+(\zeta(t), a)) \qquad \forall \zeta(t) \in \mathcal{S}, \forall a \in \mathcal{A} \tag{4.16}$$

While a value-function maps from the state space to a single value, a Q-function maps from a state-action space to a value.

Exploration For the approximation of a Q-function we can use forward exploration (section 4.3.1) and backward exploration (section 4.3.2). In contrast to the exploration runs for learning a value-function we record states *and* actions this time. If we have collected a number of successful explorations leading to the target we assign Q-values according to the following rule:

$$Q(\zeta(t), a(t)) = \begin{cases} +1 & \text{if } \zeta(t+1) \in \mathcal{S}_{target} \\ \gamma \cdot Q(\zeta(t+1), a(t+1)) & \text{else} \end{cases} \tag{4.17}$$

Here, γ is a discount factor ($\gamma \in (0, 1)$). Similarly, we can use the linear rule

$$Q(\zeta(t), a(t)) = \begin{cases} +1 & \text{if } \zeta(t+1) \in \mathcal{S}_{target} \\ Q(\zeta(t+1), a(t+1)) - \frac{1}{t_{expl}} & \text{else} \end{cases} \tag{4.18}$$

where t_{expl} is the maximal number of steps needed for an exploration run. Applying the above equations we obtain patterns

$$\langle \zeta(t), a(t) \rangle, Q(\zeta(t), a(t)) \qquad \zeta(t) \in \mathcal{S},\ a(t) \in \mathcal{A},\ Q(\zeta(t), a(t)) \in (0, 1] \tag{4.19}$$

that we can use for learning.

Learning Once we have a sufficient number of patterns we can use them in order to learn the Q-function

$$Q : \begin{cases} \mathcal{S} \times \mathcal{A} \to I\!R \\ \langle \zeta(t), a \rangle \mapsto f_{approx}(\zeta(t), a) \end{cases} \tag{4.20}$$

by means of a function approximator f_{approx}. We prefer to use neural networks (see appendix A.2) and networks of radial basis functions (see appendix A.3) for the approximation of Q. These preferences are based on the same reasons specified for the approximation of value-functions in section 4.4.3.

[2]The name Q is given in analogy to Q-learning [Watkins and Dayan 1992]

Exploitation Assuming we have learned the Q-function we want to know which action a in a given state $\zeta(t)$ leads to the best successor state $\zeta(t+1)$. This is done by heuristically generating a number of possible actions and evaluating each action's Q-value for the current state. In case of using the Q-function, the policy π is defined

$$\pi(\zeta(t)) = \arg\max_{a \in \mathcal{A}} Q(\zeta(t), a) \tag{4.21}$$

4.4.6 Policy-Function Approximation

If heuristically generating actions and computing a value (or a Q-value) for each action becomes too laborious there is another way to learn the policy: We can directly learn the policy-function that maps from a state to an action. In the following we explain the procedure for directly learning the policy-function.

Exploration For exploration we use forward exploration (section 4.3.1), backward exploration (section 4.3.2), fixed-action exploration (section 4.3.3), or supervised exploration (section 4.3.4) dependent on the initial knowledge about the task at hand. We record states *and* actions and assign actions

$$\pi(\zeta(t)) = a(t) \tag{4.22}$$

to obtain patterns

$$\langle \zeta(t_1), \zeta(t_2) \rangle, a(t_1) \qquad \zeta(t_1), \zeta(t_2) \in \mathcal{S}, \;\; a(t_1) \in \mathcal{A} \tag{4.23}$$

where $\zeta(t_1)$ is the current state and $\zeta(t_2)$ any succeeding state of the trajectory gained from exploration. If we assume the target state to be implicitly defined or if we can compute $\zeta(t_2)$ relative to $\zeta(t_1)$ (only if the state space is invariant in all dimensions) we can simplify the patterns to

$$\zeta(t), a(t) \qquad \zeta(t) \in \mathcal{S}, \;\; a(t) \in \mathcal{A} \tag{4.24}$$

and use the simplified patterns for learning. Otherwise we have to learn a more complex policy-function.

Learning Dependent on the patterns at hand we use a function approximator to learn

$$\pi(\zeta(t)) = f_{approx}(\zeta(t)) \tag{4.25}$$

approximation ↓ exploration→	forward	backward	fixed-action	supervised
value-function	5.1, 5.2	5.1, 5.7	-	-
two value-functions	5.4	-	-	-
Q-function	5.3	-	-	-
policy-function	-	-	5.6	5.5

Table 4.1: Exploration strategies and function approximations used in chapter 5.

or to learn a more complex policy-function:

$$\pi(\zeta(t), \zeta_{target}) = f_{approx}(\zeta(t), \zeta_{target}) \tag{4.26}$$

Suitable approximators include neural networks (appendix A.2) and networks of radial basis functions (appendix A.3).

Exploitation Since π is computed directly using a function approximator we simply give the current state as an input to π and obtain the action recommended. In case of a dynamic target state we have to put the target state into the policy-function to.

4.4.7 Comparison of the Different Policies

In the previous subsections we have introduced a number of different approaches to compute the policy of a task. The main difference between the individual methods proposed is on which level we apply function approximation.

While the approximation of the policy-function is straightforward it is the most complex task for function approximation because we have to estimate a direct mapping from the state space to the action space. The action space might be multi-dimensional. So the policy-function has an output completely different from the output of the value-function or the Q-function. These functions map to one dimension only. Further the policy-function might be far from smooth with a lot of local extrema. Unfortunately we cannot approximate a policy-function for all tasks. This is the main reason why we are forced to use other functions as well.

The Q-function needs heuristics to generate possible actions by the advantage that we have to approximate a less complex function that will be relatively smooth: Similar states and actions will result in similar successor states. As defined above, the Q-function has state and action as input. If even this is too complex to learn (this is the case in quite a lot of interesting tasks) we have to use a value-function supported by a projection-function. In this case we "only" have to learn a mapping from the state space to an interval and a projection-function.

Despite the differences between the individual methods introduced, they all have in common the inductive and non-incremental learning from experience and interaction.

Table 4.1 depicts what kind of exploration strategies and function approximations are used to solve the tasks following in chapter 5.

Chapter 5

Applications of Experience-Based Control

In this chapter, we present a number of problems that have been solved using experience-based learning. The tasks are related to various applications, mostly in the field of robotics and especially robot soccer. The tasks introduced here have different degrees of complexity. Some of them are obviously more complex than well known standard problems that have been solved by a number researchers. These standard problems include the cart-pole problem [Barto *et al.* 1983, Riedmiller 1993, Schaal 1997] or the mountain-car problem [Sutton 1996, Smart 2000]. Dependent on the task at hand we employ one of the methods introduced in the preceding chapter. We show that experience-based learning is a reliable approach for solving different kinds of tasks. For each task, we describe in detail what the situation of the experiments was, what algorithms were applied, and what was the result.

5.1 An Abstract Navigation Task

Before we start to apply experience-based learning techniques to more concrete tasks we begin with an abstract task. The objective, here, is to move a mobile object towards a certain target position in 2D. The continuous state space is given by

$$\mathcal{S} = [-5.0m, +5.0m] \times [-5.0m, +5.0m] \tag{5.1}$$

The target space is given by

$$\mathcal{S}_{target} = \{\zeta \in \mathcal{S} \mid \delta_0(\zeta) \leq 0.5m\} \tag{5.2}$$

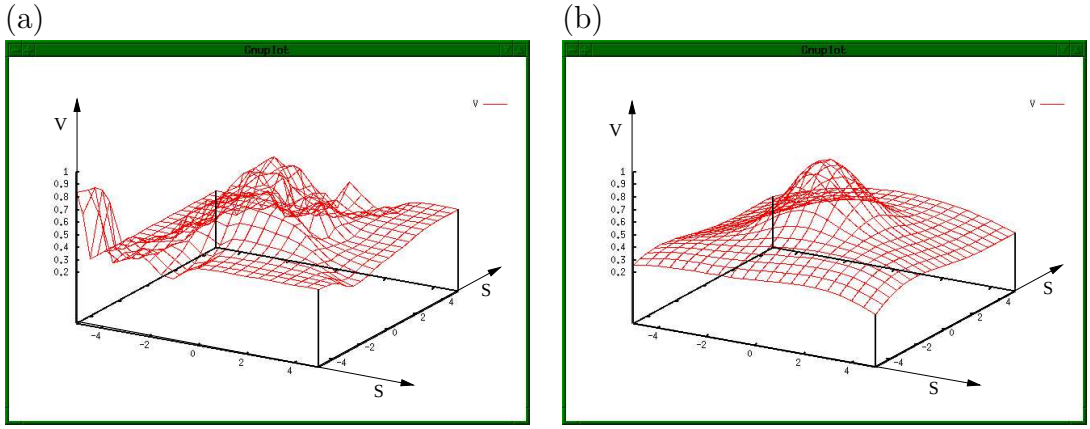


Figure 5.1: Value-functions obtained from learning. Both functions are trained with data from 1000 runs of forward exploration. 90 explorations were successful. Subfigure (a): A trained multi-layer perceptron (2-8-4-1 topology). Subfigure (b): A trained RBF-network (5 kernels).

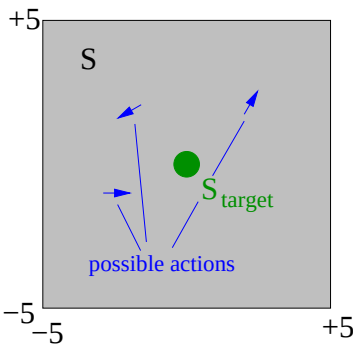


Figure 5.2: State space and possible actions of the abstract navigation task.

where $\delta_0(\zeta)$ is the length of ζ relative to the origin of the state space (using the Euclidean metric). Figure 5.2 shows the scenario. In each discrete time step, the object is moved $1m$. The continuous action space contains the available directions of the movement and is defined

$$\mathcal{A} = [0, 360) \tag{5.3}$$

where an action denotes the direction in degrees. Obviously this kind of movement is a strong simplification of real movements. Further this task can easily be solved using a hand-coded implementation. But this task is a suitable testbed for the experience-based learning methods introduced. Action space, state space, and transfer function can easily be changed to perform more difficult tasks.

Exploration For the exploration of the state space we use forward exploration (section 4.3.1) and backward exploration (section 4.3.2). For this task, the projection-functions $\mathcal{P}^+$ and $\mathcal{P}^-$ are hand-coded. We perform explorations with 1000 (100) runs for forward (backward) exploration. In addition we perform a backward exploration with one run only. The states of successful explorations are assigned values according to

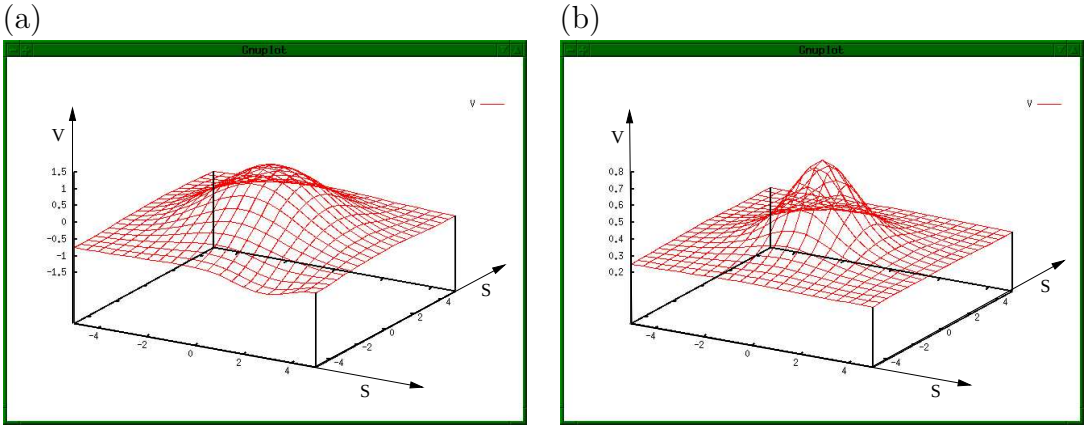


Figure 5.3: Value-functions obtained from learning. Subfigure (a): A trained RBF-network (5 kernels). Only one exploration run was performed. Subfigure (b): A trained RBF-network (5 kernels). 100 exploration runs were performed.

$$\mathcal{V}(\zeta(t)) = \begin{cases} +1 & \textit{if } \zeta(t) \in \mathcal{S}_{target} \\ \gamma \cdot \mathcal{V}(\zeta(t+1)) & \textit{else} \end{cases} \tag{5.4}$$

in order to learn a value-function $\mathcal{V}$ (see section 4.4.3). The length of an exploration is limited to 20 steps. From 1000 forward explorations performed only 90 were successful. Naturally, all of the 100 backward explorations can be used in order to generate patterns $\zeta(t), \mathcal{V}(\zeta(t))$. γ was set to 0.9. For validation by means of early stopping a second set of 1000 forward explorations was performed. 105 of the runs of this set were successful.

Learning For learning the patterns we use multi-layer neural networks (appendix A.2) and networks of radial basis functions (appendix A.3). The neural network has a relatively simple topology (2-8-4-1), while the RBF-networks have 5 kernels. The neural network was trained using the validation set of 105 successful explorations for early stopping [Sarle 1995] (the n++ software, section 5.8.1, was used). The RBF-networks were trained using features for automatic optimization as described in the software section 5.8.2. The resulting value-functions are depicted in figures 5.1 and 5.3.

Exploitation As written in section 4.4.3, equation (4.9) we choose the action promising the highest value for the successor state for execution. In our experiments we define a set of 1000 randomly chosen start states and use them for exploitation with the different value-functions. Table 5.1 depicts the percentage of successful exploitations and the average number of steps to the target for four different value functions.

Value-function	successful exploitations	average time steps per exploitation
MLP 2-8-4-1 forw1000	86.4%	6.274
RBF 5kernels forw1000	100.0%	4.473
RBF 5kernels back001	100.0%	4.493
RBF 5kernels back100	100.0%	4.279

Table 5.1: Percentage of successful exploitations and average number of steps for different value-functions.

Herein *MLP 2-8-4-1 forw1000* means the value-function approximated by the multi-layer neural network based on 1000 forward explorations, *RBF 5kernels forw1000* means the value-function approximated by an RBF-network of 5 kernels based on 1000 forward explorations, *RBF 5kernels back001* means the value-function approximated by an RBF-network of 5 kernels based on one backward exploration, and *RBF 5kernels back100* means the value-function approximated by an RBF-network of 5 kernels based on 100 backward explorations.

As we can see, the RBF-networks performs a lot better than the multi-layer neural network. This is not surprising since the RBF-networks use Gaussian kernels and these are appropriate means to represent the nonlinear distance metric of this task because their natural shape is similar to the desired value-function.

The value-function of the multi-layer neural network (figure 5.1(a)) is not strictly monotonic increasing towards the target. The reason for that is probably a lack of training data. It depends on the validation data how long the network is trained. If the explorations of the validation data are totally different from those of the training data the result might be poor.

Obviously the performance of the RBF-networks is best using backward exploration. This makes sense since all backward explorations can effectively be used for generating training data. Inherently, more training data improve the performance of the learned value-function.

5.2 The Corridor-Following-Task

The corridor-following task involves learning to control a robot in order to navigate through a corridor towards a target point at the end of the corridor. The state space $\mathcal{S}$ consists of the distance to the end of the corridor, δ_{target} (in meters) the orientation of the robot relative to the corridor, $\Delta\varphi$ (in rad $(-180, 180]$) and the coordinate of the robot (in terms of the perpendicular of the corridor) in relation the middle of the corridor, δ_y (in meters):

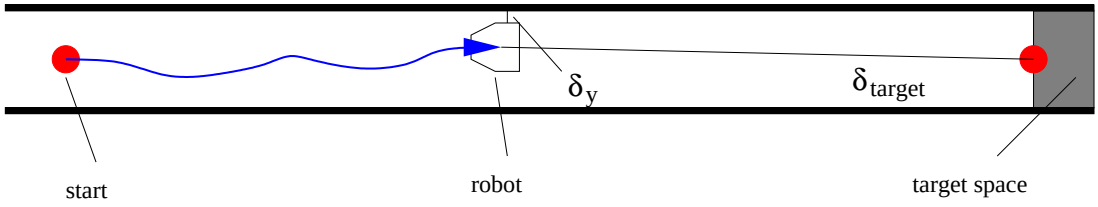


Figure 5.4: The corridor following task: A robot has to navigate from a start state to a target state in a corridor.

$$\zeta = \langle \delta_{target}, \Delta\varphi, \delta_y \rangle \tag{5.5}$$

Figure 5.4 shows a possible scenario. In our experiments the objective of the policy π is to minimize the amount of time the robot needs to reach the target state by choosing an appropriate rotational velocity (in degree per second) while the translational velocity of the robot is fixed. The target space $\mathcal{S}_{target}$ is reached if the robot is at the end of the corridor (independent of $\Delta\varphi$ and δ_y).

An only slightly different task has been defined by [Smart and Kaelbling 2000] and has been used for the evaluation of their reinforcement learning algorithms. In this work, the robot uses a laser range-finder to localize itself on the corridor. Around 60 training runs were performed.

In our case the distance between start and target state is 8 meters while the width of the corridor is 0.8 meters and the diameter of the robot is 0.4 meters. This means the robot has very little space for moving at all.

Exploration During exploration we chose a random rotational velocity $V_{rot} \in \mathrm{N}(0, 30)$ (N is the Gaussian distribution). The maximal absolute value for V_{rot} is 180. We change the velocity after a random time. Thus the action space $\mathcal{A}$ is $[-180, +180]$. The robot starts at $\zeta_{start} = (8.0, 0.0, 0.0)$. We use forward exploration (section 4.3.1) in order to learn a value-function for the given state space. Reaching the target or colliding with the wall are terminal states. If the robot collides with the wall we ignore all data of the exploration. Otherwise we assign values to the states recorded according to

$$\mathcal{V}(\zeta(t)) = \begin{cases} +1 & \textit{if } \text{robot has reached end of corridor} \\ \gamma \cdot \mathcal{V}(\zeta(t+1)) & \textit{else} \end{cases} \tag{5.6}$$

We perform ten forward explorations of which *only two* were successful. The patterns obtained are used for learning the value-function.

(a) (b)

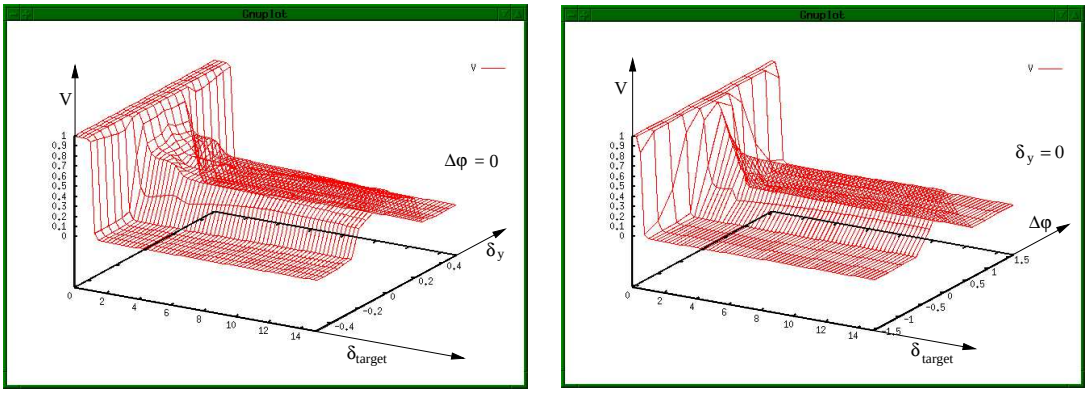


Figure 5.5: The value-function after 10 exploration runs. Subfigure (a): $\mathcal{V}$ dependent on δ_{target} and δ_y . Subfigure (b): $\mathcal{V}$ dependent on δ_{target} and $\Delta\varphi$.

Learning To learn the patterns $\zeta(t), \mathcal{V}(\zeta(t))$ we employ multi-layer neural networks (appendix A.2) because of their global generalization behavior which is, in contrast to RBF-networks, not limited to the local neighborhood of training data but covers the whole state space. The network was trained using the RPROP algorithm [Riedmiller and Braun 1993], and early stopping [Sarle 1995]. A simple 3-8-1 topology was sufficient for a reasonable performance of the network. The n++ software described in section 5.8.1 was used. The resulting value-function is depicted in figure 5.5(a) dependent on δ_{target} and δ_y and in figure 5.5(b) dependent on δ_{target} and $\Delta\varphi$.

Exploitation We choose the action promising the highest value for the successor state for execution (according to section 4.4.3, equation (4.9)). During exploitation we choose actions from

$$V_{rot} \in \{-180, -170, \ldots, 0, \ldots, +170, +180\} \tag{5.7}$$

and evaluate all 37 possible successor states at a frequency of 10 Hz. In practice we achieve a success rate of 100% in reaching the target while in the ten runs of exploration the success rate was 20% only. The optimal time to reach the target (driving almost straight) is 10.9 seconds. The time needed by our policy is 10.9 seconds as well. If we add Gaussian noise on our chosen action

$$V_{rot} = \pi(\zeta(t)) + \mathrm{N}(0, 60) \tag{5.8}$$

the time needed is 11.1 seconds only. Due to its acceleration behavior the robot never reaches a high rotational velocity. The policy automatically controls the robot in order to keep it in the middle of the corridor.

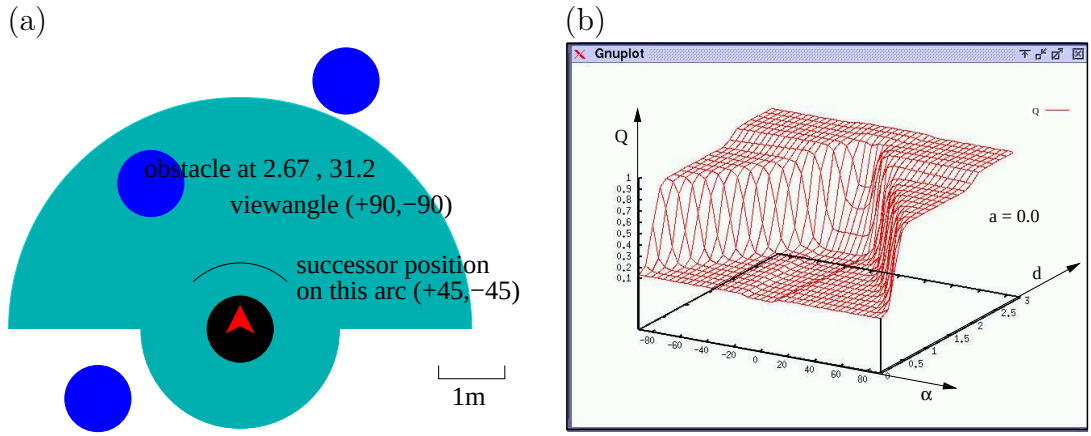


Figure 5.6: Subfigure (a): The simulated robot perceives objects up to a distance of $1m$ all around it and up to $4m$ in its view-angle. Subfigure (b): The Q-function of obstacle avoidance for a fixed action $a = 0.0$.

As we can see, the value-function is strictly monotonic increasing towards the target and not of a rugged shape. This underlines its quality.

However, the corridor-following task is still easy to solve by hand-coded implementations. Furthermore, the optimal value function is strictly monotonic increasing in all dimensions of the state space what proves this task to be a simple one. But the fact that only two successful exploration runs are enough to learn a value-function that enables the robot to reach its target in minimal time earns attention.

Robot Platform

The robot used for the corridor-following task was a Pioneer I robot [Pioneer 1998]. To generate training data it was accurately simulated by means of neural networks (described in section 3.5.1). Details can be found in [Buck *et al.* 2002c] too.

5.3 Obstacle Avoidance

In this task we want a robot to learn to avoid obstacles that occupy spaces in the robot's local environment. Experiences are made from collisions with obstacles during exploration. The robot and all the obstacles have circular floor plan and a diameter of 1 meter. The robot moves 1 meter forward per discrete time step. After every meter it decides whether to turn by maximal 45 degrees to the left or right. Thus the action space is defined

$$\mathcal{A} = [-45, +45] \tag{5.9}$$

The simulated robot perceives objects up to a distance of 1 meter all around it and up to 4 meters if they are in its view (see figure 5.6(a)). The robot gets information on the distance and the angle of the obstacles. Its view ranges up to 90 degrees to the left and to the right. The state of the robot regarding *one* certain obstacle O is defined

$$\zeta_O = \langle d, \alpha \rangle \tag{5.10}$$

with d (α) being the distance to (angle to) obstacle O. d is measured in meters and α is measured in degrees; $\alpha \in (-180, 180]$.

In this application, we choose to approximate a Q-function because the state space is two-dimensional only. The combined state-action space (the input of the Q-function) has only three dimensions. For exploration we train the robot with static obstacles while during exploitation we will try the Q-function for static *and* dynamic obstacles. The advantage of this procedure is that the size of the state space can be kept moderate for learning.

Exploration During exploration we put the robot on a random position in a room of 40×40 square meters. 100 obstacles are placed in the room randomly. Figure 5.7 depicts a part of the scenario. Then the robot chooses random actions, $a \in \mathcal{A}$. Patterns are generated by regarding sequences of k actions. If the robot moves k steps without a collision with obstacle O we assign values to all preceding combinations of states and actions according to

$$Q(\zeta_O(t), a(t)) = 1 \quad \forall t \in \{1, \ldots, k\} \tag{5.11}$$

In the other case, if the robot collides with obstacle O in the ith step, we assign values according to

$$Q(\zeta_O(t), a(t)) = \frac{i - t}{k} \quad \forall t \in \{1, \ldots, i\} \tag{5.12}$$

Patterns can be generated only for obstacles that are perceived by the robot.

Learning For the approximation of the Q-function we use multi-layer neural networks (appendix A.2) because of their minimal computational amount during exploitation. We need to compute the network's values several times per cycle. The network was trained using the RPROP algorithm

[Riedmiller and Braun 1993], and early stopping [Sarle 1995] (the n++ software, section 5.8.1, was used.) A simple 3-5-1 topology performed good enough for this task. The trained Q-function resulting from the patterns $\langle \zeta_O(t), a(t) \rangle, Q(\zeta_O(t), a(t))$ is depicted in figure 5.6(b) for the action $a = 0.0$.

Exploitation The policies used for exploitation are given in the following subsections dependent on the number of obstacles.

5.3.1 One Obstacle

If we have to take one obstacle into account the policy given in section 4.4.5, equation (4.21) can be used. The Q-function learned (figure 5.6(b)) shows that we get low Q-values for obstacles close to the robot (low values for d) and that we get bad values even for distant obstacles if they are in the robot's moving direction.

5.3.2 Multiple Obstacles

Now we regard the same scenario but with multiple obstacles. The robot has to go through a room of 40×40 square meters with 250 moving obstacles this time. The robot *must* move 1 meter per simulation cycle but can choose an angle between -45 and $+45$ degrees. Obstacles move at 1 meter per cycle as well. They move to randomly defined targets.

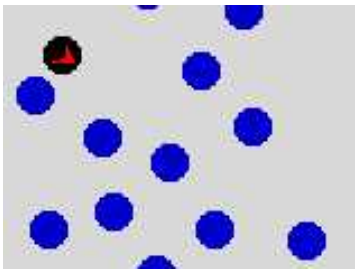


Figure 5.7: Obstacle avoidance: An environment containing for exploration and exploitation with multiple robots.

Since the Q-function was trained for one obstacle only but we have multiple obstacles now, we need to define a new policy-function. This policy chooses the action promising the maximal Q-value for the current state minimizing the Q-value over all obstacles in the range of the robot's perception. In the following equation $\mathcal{O}$ denotes the set of obstacles perceivable by the robot.

$$\pi_{O \in \mathcal{O}}(\zeta_O(t)) = \arg\max_{a \in \mathcal{A}} [\min_{O \in \mathcal{O}} Q(\zeta_O(t), a)] \tag{5.13}$$

In our exploitation runs we generate 40 possible actions and evaluate all their Q-values for the current states of all obstacles perceived. That leads to a success rate of 98.8% for static obstacles and to a success rate of 88.6% for dynamic obstacles. The tests include around 3000 trials for each of both exploitations. Herein success means that the robot does not collide with any obstacle. Regarding the lower rate

Figure 5.8: The dribble-task: The forward (with ball) must pass by the defender.

for dynamic obstacles one must consider that there are cases in which the robot cannot avoid all obstacles by any action. The robot can even be surrounded by obstacles in this crowded environment (see figure 5.7). Further exploration was done with static obstacles only. The success rate during exploration was close to zero.

A short animation related to this experiments can be found at
http://www9.in.tum.de/archive/agilo/ObstacleAvoidance.gif

Robot Simulation

For this task we used a virtual robot. The position of the robot in 2D at time t, $p(t)$ is computed by

$$p(t) = p(t-1) + \begin{pmatrix} cos(\varphi(t)) \\ sin(\varphi(t)) \end{pmatrix} \tag{5.14}$$

where $\varphi(t)$ is the current orientation of the robot normed to the interval $[0, 360)$. It is updated by

$$\varphi(t) = ||\varphi(t-1) + a(t-1)||_{[0,360)} \tag{5.15}$$

5.4 The Dribble-Task

All the tasks described in the preceding sections have at most three-dimensional state space. Most of those tasks can be solved by hand-coded controllers. The task introduced in this section deals with a fast changing and adversarial real

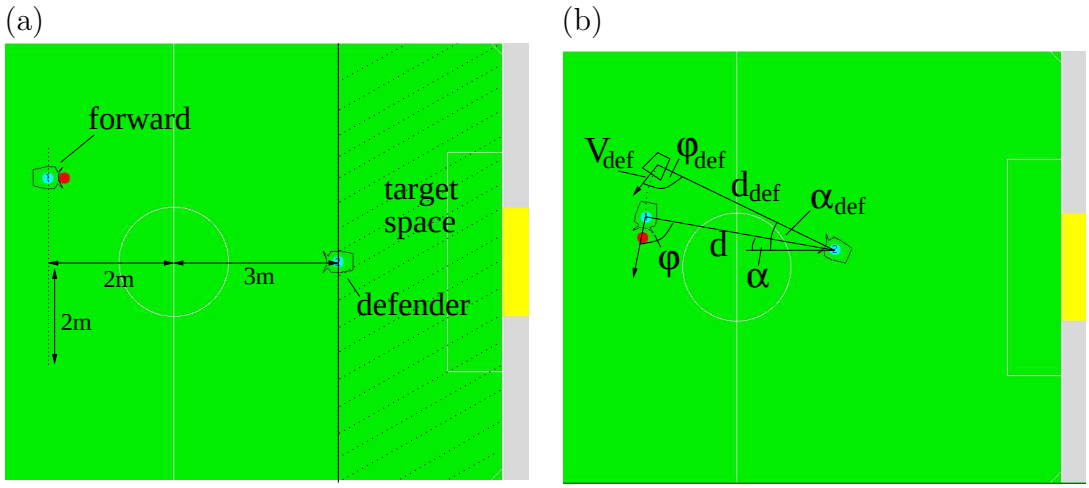


Figure 5.9: The dribble-task. Subfigure (a): The initial setting of the task. The forward has to dribble around the defender to reach the target space. Subfigure (b): Some features of the state space.

world environment and is therefore much more complex. As we will see later, an optimal value-function will not be strictly monotonic increasing (or decreasing) in all dimensions of the state space. Learning to dribble in autonomous robot soccer provides a good testbed for a multi-dimensional robot control problem.

Let us consider the following problem: A soccer robot in possession of the ball has to dribble around an opponent defender. The opponent robot wants to hinder it and tries to get the ball (figure 5.8). Especially the adversarial environment (namely the defender) makes this task a very difficult one. An appropriate state space for this environment must include relative distances, angles, and velocities of the robots. Hence the dimension of the state space will be much more than three.

In human soccer, players can quickly change their direction and, not seldom, try to deke opponent players by performing fake-out motions. They move into one direction for a moment just to make the opponent move in that direction too. Exploiting the delay of the opponent's perception as well as its physical inertia players quickly change their direction. We try to learn such a behavior for robot soccer using value-function approximation (section 4.4.3).

Let us now introduce the scenario of the dribble-task [Buck *et al.* 2002b]: There is one robot (called *forward*) that has a ball in its guiding device. A second robot (called *defender*) tries to get the ball from the forward using a fixed and deterministic policy. At the beginning the defender is placed at $(3m, 0m)$ and the forward (with ball) at $(-2m, y)$ with $y \in [-2m, +2m]$ and $(0m, 0m)$ being the center of a robot soccer field. The forward's target space is defined by all points *behind* the defender in terms of the x-coordinate.

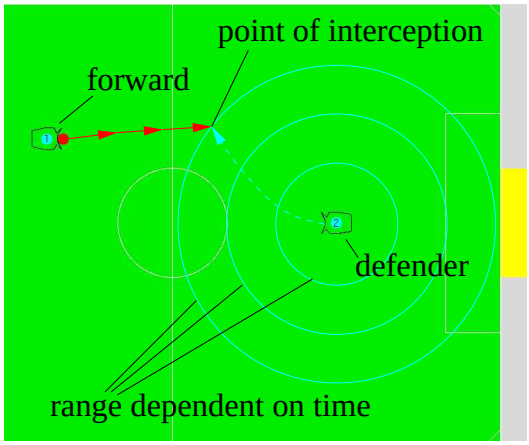


Figure 5.10: The point of interception for the defender.

The state space consists of the features d (the distance between forward and defender), α (the angle between forward and defender), φ (the forward's orientation relative to α), V_{tr} and V_{rot} (the forward's translational and rotational velocities), d_{def} (the distance between forward and defender observed by the defender), α_{def} (α observed by the defender), φ_{def} (φ observed by the defender), V_{def} (V_{tr} observed by the defender), and V_{max} (the maximal velocity of the defender). Figure 5.9 depicts the task (subfigure(a)) and some components of the state space (subfigure (b)).

The policy of the defender is to intercept the ball at the point where the direction of its velocity-vector touches the range of the defender (as a function of time). Figure 5.10 illustrates the behavior. The policy of the defender is assumed to be deterministic. However, its movement must not be deterministic.

The policy of the forward has to be learned. Actually, for the dribble-task *two* subtasks have to be solved:

(1) A model of the defender's behavior has to be generated because the policy depends on the defender's behavior.

(2) The forward must learn a policy to pass by the defender.

In this work, we concentrate on the learning of motion control (subtask (2) in the above context), and therefore neglect subtask (1) although this is a tough task too.

One can see, the learning of a policy for the dribble-task obviously is much more complex than, for instance, for the corridor-following task:

- A reasonable state space will have about *ten dimensions*.
- The value-function is *not strictly monotonic increasing (decreasing) in all dimensions* of the state space.
- Any successor state depends not only on the behavior of the controlled robot but on the *behavior of the opponent* too (adversarial environment).

According to the subtasks (1) and (2) we split the state space $\mathcal{S}$ into a part dependent on the perception of the forward, $\mathcal{S}_{forw}$ and a part dependent on the perception of the defender, $\mathcal{S}_{def}$:

$$\mathcal{S} = \mathcal{S}_{forw} \times \mathcal{S}_{def} \tag{5.16}$$

The part of the state concerning the perception of the forward consists of

$$\zeta_{forw} = \langle d, \alpha, \varphi, V_{tr}, V_{rot} \rangle \tag{5.17}$$

while the part of the state concerning the perception of the defender consists of

$$\zeta_{def} = \langle d_{def}, \alpha_{def}, \varphi_{def}, V_{def}, V_{max} \rangle \tag{5.18}$$

As written above deking is based on delays in perception and motion. Further the perception of the defender, ζ_{def} definitely has something to do with some former perception of the forward. Since we know the defender will compute a point where it can intercept the ball (figure 5.10) and will simply go there we need to construct a model of its perception and motion abilities (subtask (1)). This can be done by online estimating delay parameters and computing the defender's maximal velocity. Since this would be another chapter (obtaining a model for hidden state is not a goal of this work) we provide reasonable values of $\mathcal{S}_{def}$ to the forward robot in the following experiments in order to learn a policy. We want to learn the policy by means of value-function approximation. For the computation of the value of a state we need the state as observed by the forward, $\zeta(t)_{forw}$ and the provided information about the defender, $\zeta(t)_{def}$. The value-function is defined

$$\mathcal{V} : \begin{cases} \mathcal{S} \to I\!R \\ \zeta(t) \mapsto \mathcal{V}(\zeta(t)) \end{cases} \tag{5.19}$$

The forward-projection-function $\mathcal{P}^+$ (which is part of the input for $\mathcal{V}$ ($\mathcal{V}^+$ and $\mathcal{V}^-$ respectively) according to equations (4.9) and (4.13)) maps to $\mathcal{S}_{forw}$:

$$\mathcal{P}^+ : \begin{cases} \mathcal{S}_{forw} \times \mathcal{S}_{def} \times \mathcal{A} \to \mathcal{S}_{forw} \\ \langle \zeta_{forw}(t), \zeta_{def}(t) \rangle, a(t) \mapsto \zeta_{forw}(t+1) \end{cases} \tag{5.20}$$

Analogously to the split of $\mathcal{S}$ we split our projection-function $\mathcal{P}^+$ as well:

$$\mathcal{P}^+ = \mathcal{M} \circ (\mathcal{P}^+_{forw} \times \mathcal{P}^+_{def}) \tag{5.21}$$

where $\mathcal{P}^+_{forw}$ is a projection-function for the change in state of $\mathcal{S}_{forw}$ (only depending on the robot's action because the defender's behavior is deterministic

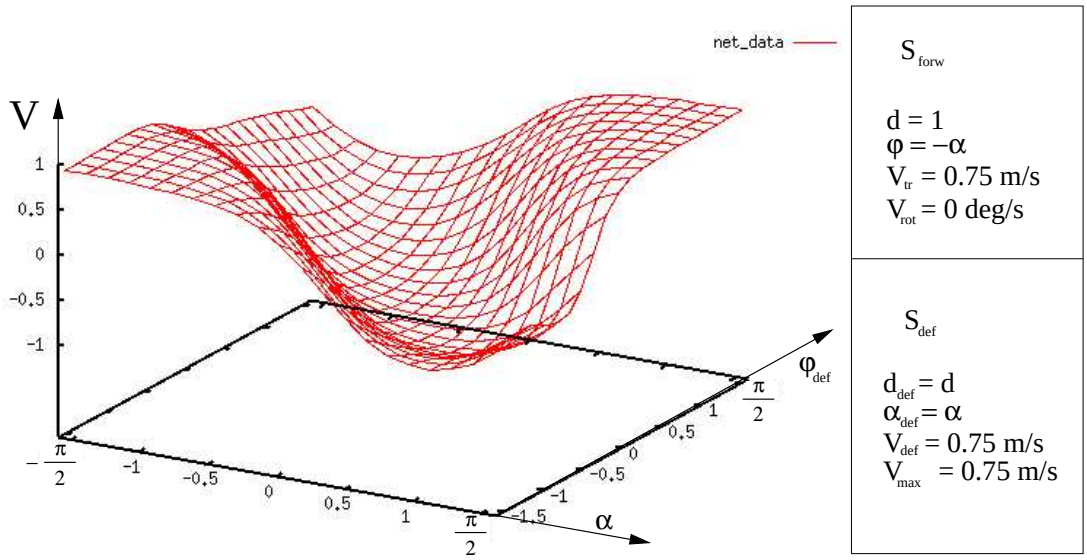


Figure 5.11: Generalization of the trained neural network. The graphic shows the output of the network computing the value-function with fixed input in all but two dimensions.

and depends on the state only). $\mathcal{P}^+_{def}$ is a projection-function for the change in state of $\mathcal{S}_{def}$, and $\mathcal{M}$ is a merging-function:

$$\mathcal{P}^+_{forw} : \begin{cases} \mathcal{S}_{forw} \times \mathcal{A} \rightarrow \mathcal{S}_{forw} \\ \zeta_{forw}(t), a(t) \mapsto \zeta_{forw}(t)_{succ} \end{cases} \tag{5.22}$$

$$\mathcal{P}^+{}_{def} : \begin{cases} \mathcal{S}_{def} \rightarrow I\!R^2 \\ \zeta_{def}(t) \mapsto \langle \Delta x, \Delta y \rangle \end{cases} \tag{5.23}$$

$$\mathcal{M} : \begin{cases} \mathcal{S}_{forw} \times I\!R^2 \rightarrow \mathcal{S}_{forw} \\ \zeta_{forw}(t)_{succ}, \langle \Delta x, \Delta y \rangle \mapsto \zeta_{forw}(t+1) \end{cases} \tag{5.24}$$

$\mathcal{P}^+_{forw}$ computes the movement of the forward in the part of the state space perceivable by the forward ($\mathcal{S}_{forw}$) dependent on the action chosen, $a(t)$. $\mathcal{P}^+_{def}$ computes the change in the position of the defender, $\langle \Delta x, \Delta y \rangle$ (relative to its previous position). This change depends on the part of the state space perceivable by the defender, $\mathcal{S}_{def}$, only. The merging-function $\mathcal{M}$ maps the movement of the forward and the change in position of the defender to a successor state in $\mathcal{S}_{forw}$ (perceivable by the forward) that can be used as a part of the input for the value-function (equation (5.19)).

The action space of the dribble-task is $[-180, +180]$ where an action denotes the rotational velocity of the forward ($a = V_{rot}$). The translational velocity of the forward is assumed to be $0.75m/s$, constantly.

Exploration During exploration the robot performs random actions each for a random time. This means the robot drives into different directions and for a certain time in each direction. An initial hand-coded policy is also used: In this policy the robot is given a direction and at a randomly chosen point the robot suddenly changes its direction rapidly. We perform a set of 500 explorations with an average duration of about 6 seconds (this means we get sequences of 60 states at a frequency of 10 Hz). Further 250 explorations were performed to obtain validation data for an early stopping of the function approximation. Altogether that leads to around 45000 patterns obtained in less than two hours of simulation. The success rate for passing by the defender is less than 10% during exploration.

The states visited during exploration are assigned values according to

$$\mathcal{V}^+(\zeta(t)) = \begin{cases} +1 \;\; \textit{if } \zeta(t) \in \mathcal{S}_{target} \\ \gamma \cdot \mathcal{V}^+(\zeta(t+1)) \;\; \textit{else} \end{cases} \tag{5.25}$$

$$\mathcal{V}^-(\zeta(t)) = \begin{cases} -1 \;\; \textit{if defender touches the forward} \\ \gamma \cdot \mathcal{V}^-(\zeta(t+1)) \;\; \textit{else} \end{cases} \tag{5.26}$$

$\mathcal{V}^-(\zeta(t))$ is set to -1 if the defender touches the forward or the forward exceeds the time limit for exploration. $\mathcal{V}^+(\zeta(t))$ is set to $+1$ if the forward reaches the target space. Otherwise the discount-factor γ is used in order to assign values to the preceding states.

Learning The patterns $\langle \zeta(t), \mathcal{V}^{+/-}(\zeta(t)) \rangle$ are trained using neural networks with 10-8-1 topology in order to learn the value-function $\mathcal{V}$. One hidden layer was enough to achieve a good performance. We choose neural networks because of their global generalization behavior. This behavior is not limited to the local neighborhood of the training data. Additional features for learning like RPROP [Riedmiller and Braun 1993] and early stopping [Sarle 1995] were used (see appendix A.2 and section 5.8.1 for detailed descriptions).

Exploitation During exploitation actions are chosen from

$$a \in \{-120, 0, 120\} \tag{5.27}$$

which is a quite coarse selection. The policy of equation (4.13) was used for choosing actions ($\mathcal{V} = \mathcal{V}^+ + \mathcal{V}^-$). Surprisingly, the success rate in the exploitation

phase was 100% although the action was chosen out of three possibilities only. These results are based on 250 runs and the experimental setup described above. Figure 5.11 shows two dimensions of our trained value-function with the other dimensions set by fixed rules. One can see the smooth surface as an indication for good generalization.

An animation (gif,870KB) of a typical dribble scene can be viewed at *http://www9.in.tum.de/archive/agilo/DribbleTask.gif*.

Robot Platform

For the forward we used a Pioneer I robot [Pioneer 1998]. Using means of neural networks (described in section 3.5.1) it was accurately simulated to generate training data. Further information about the simulation can be found in [Buck *et al.* 2002c] too.

For the defender we use a Pioneer I robot too. It was simulated the same way the forward was. The state as perceived by the defender, $\zeta_{def}(t)$ was computed as a function of a sequence of states perceived by the forward in the past:

$$\zeta_{def}(t) = f(\zeta_{forw}(t - 100ms), \ldots, \zeta_{forw}(t - 1000ms)) \tag{5.28}$$

Herein the delay for the position is very small while the delay for the components of $\zeta_{def}(t)$ concerning velocities is up to one second.

5.5 Robot Navigation

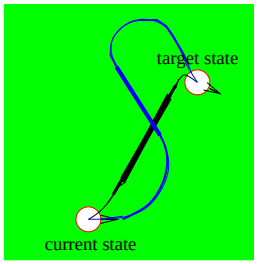


Figure 5.13: Different ways to reach the target.

The basic component of a navigation system is a controller that enables the robot to achieve specified dynamic states quickly. Such a controller receives the target state (for example, the next state on a planned path) of a robot and returns low level commands that transform the current state into the target state.

As shown in figure 5.13 there are different ways to solve this problem (the width of the trajectories indicates the robot's translational velocity). To arrive at the target state different trajectories are possible. What makes this task difficult is that the dynamic state of a robot contains not only its position but its orientation and its velocity too [Buck *et al.* 2001b]. In certain applications the target state of a robot includes a high velocity. Therefore a robot cannot drive directly towards the target and then turn on the spot. In this

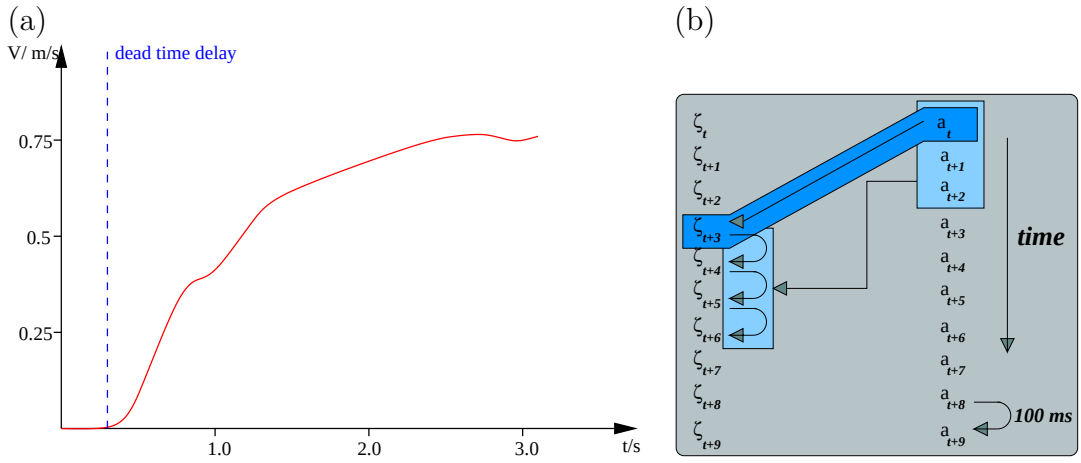


Figure 5.12: Subfigure (a): The acceleration curve of a Pioneer I robot resulting from $a = (0, 0)$ for any $t < 0$ and $a = (0.75, 0)$ for $t \geq 0$. The dead time delay is about $300ms$. Subfigure (b): Overcoming the dead time delay by assigning a to the change in state taking place $300ms$ ahead.

section, we consider no obstacles (this would make the task much more complex). Real path planning is considered in section 5.6.

The dynamic state of a Pioneer I robot [Pioneer 1998] can be summarized as a quintuple

$$\zeta = \langle x, y, \varphi, V_{tr}, V_{rot} \rangle \tag{5.29}$$

where x and y are coordinates in a global system, φ is the orientation of the robot and V_{tr} (V_{rot}) are the translational (rotational) velocities. Using the Saphira software [Konolige *et al.* 1997] one can set a command (=action)

$$a = \langle V_{tr}, V_{rot} \rangle \tag{5.30}$$

at the frequency of 10 Hz where $V_{tr}(V_{rot})$ denotes the target velocity in meters per second (degrees per second).

The question for the low-level-controller now is, how to set V_{tr} and V_{rot} given a current state, $\zeta(t)$ and a target state, ζ_{target} in order to reach the target state precisely and quickly. We have measured a time delay of around $300ms$ from the setting of the action a until the robot executes the command (see fig. 5.12(a) and section 3.5.1). In order to take this delay into account we assign commands $a(t)$ to the change in state $\zeta(t+3) \rightarrow \zeta(t+4)$ as depicted in figure 5.12(b). In the remainder of this section $\zeta(t), \zeta(t+1)$ and $a(t)$ denote a change in state and the respective command causing it.

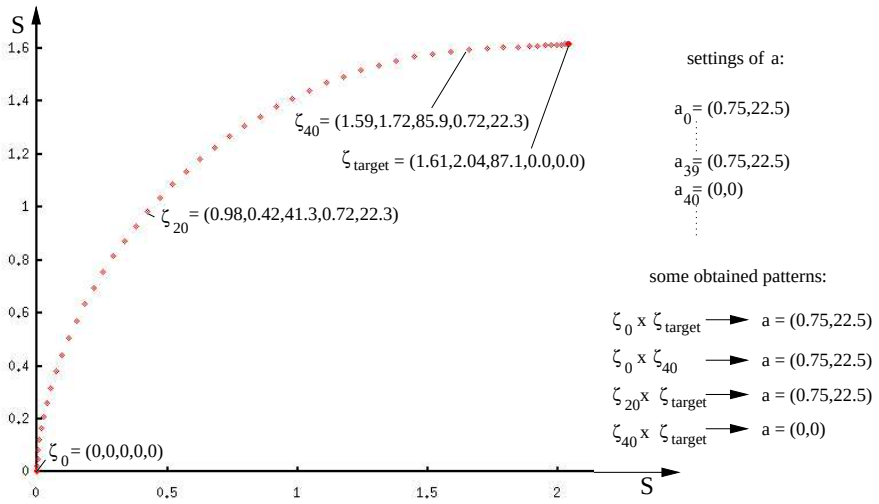


Figure 5.14: Driving 4 seconds with $a = (0.75, 22.5)$ and thereafter with $a = (0, 0)$ leads to numerous different patterns. The states from $\zeta(0) = (0, 0, 0, 0, 0)$ to ζ_{target} and 4 example patterns are depicted above.

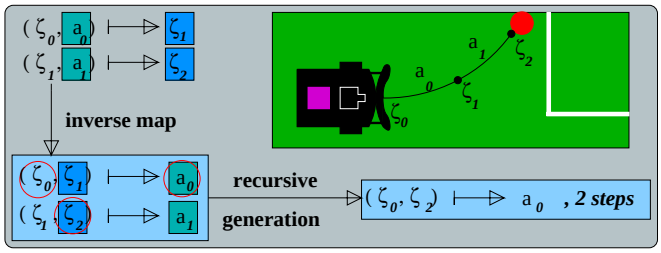


Figure 5.15: Recursive generation of training patterns for the neural controller: Two successive changes in state observed are inverted and combined to one new pattern.

In this navigation task, we want to learn a mapping from a start state and a target state to an action according to section 4.4.6 (Policy-Function Approximation). Thus the input of the policy-function depends on the robot's current state and the robot's target state.

Exploration To collect training data we do several runs where we set the action a to certain values by performing supervised exploration (section 4.3.4). We always try to drive fast and not to set actions canceling out each other. For example, driving with only half speed on a straight line will teach the controller to do so even if it could reach the target state faster. But decreasing the robot's velocity on a straight line will teach the controller to do so. In parallel to setting actions we record the changes in state resulting. Out of it we get a huge number of patterns

$$\langle \zeta(t), a(t) \rangle, \zeta(t+1) \tag{5.31}$$

These patterns are inverted to

(a) (b)

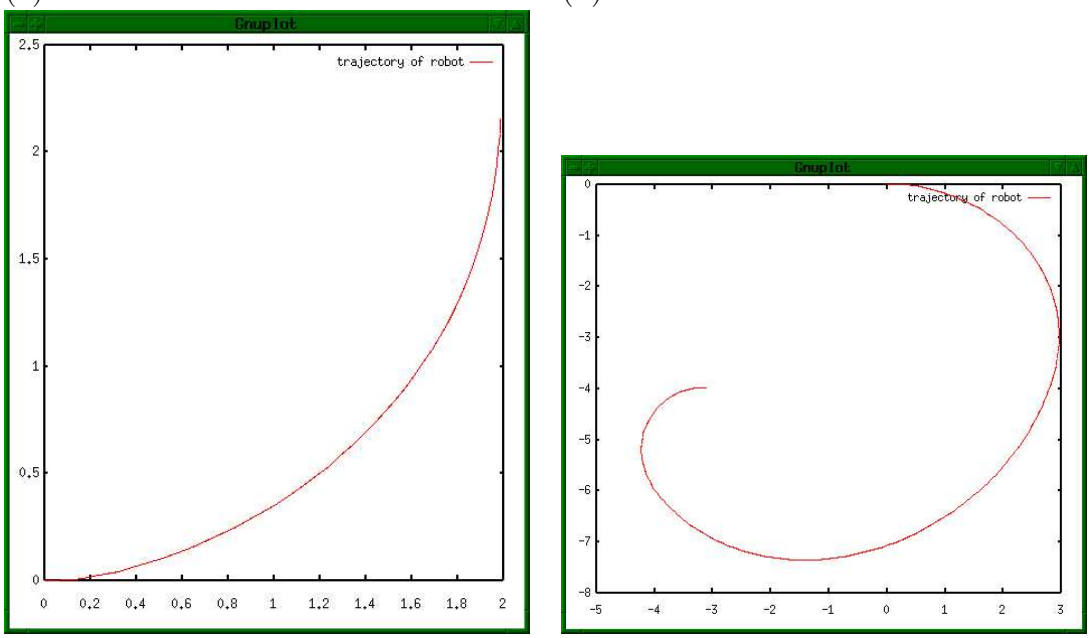


Figure 5.16: Subfigure (a): The trajectory of the robot for $\zeta(t) = \langle 0.0, 0.0, 0.0, 0.0, 0.0\rangle$ and $\zeta_{target} = \langle 2.0, 2.0, 90.0, 0.0, 0.0\rangle$. Subfigure (b): The trajectory of the robot for $\zeta(t) = \langle 0.0, 0.0, 0.0, 0.0, 0.0\rangle$ and $\zeta_{target} = \langle -3.0, -4.0, 0.0, 0.0, 0.0\rangle$.

$$\langle \zeta(t), \zeta(t+1)\rangle, a(t) \tag{5.32}$$

Successive patterns

$$\langle \zeta(t), \zeta(t+1)\rangle, a(t) \text{ and } \langle \zeta(t+1), \zeta(t+2)\rangle, a(t+1) \tag{5.33}$$

can recursively be combined to

$$\langle \zeta(t), \zeta(t+2)\rangle, a(t) \tag{5.34}$$

as illustrated in figure 5.15. From one simple trajectory we get lots of useful training patterns as we can see in figure 5.14. Patterns are created not only from start state and target state but from any two consecutive states or patterns of the trajectory. Recapitulatory, we want to learn the dynamical driving behavior of a Pioneer I robot resulting from specified sequences of actions and exploit this knowledge for navigation.

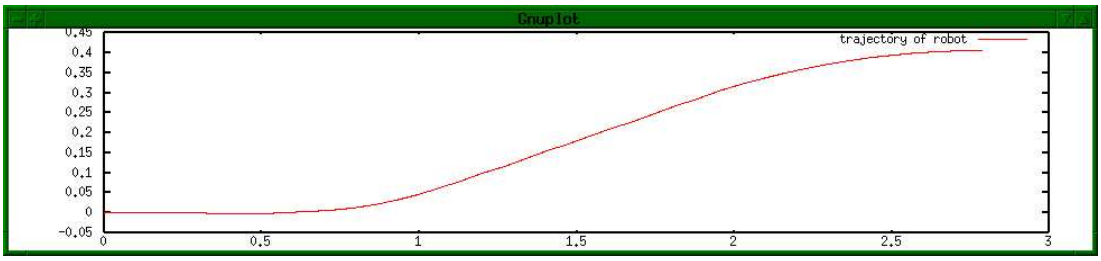


Figure 5.17: The trajectory of the robot for $\zeta(t) = \langle 0.0, 0.0, 0.0, 0.0, 0.0\rangle$ and $\zeta_{target} = \langle 2.85, 0.40, 90.0, 0.0, 0.0\rangle$.

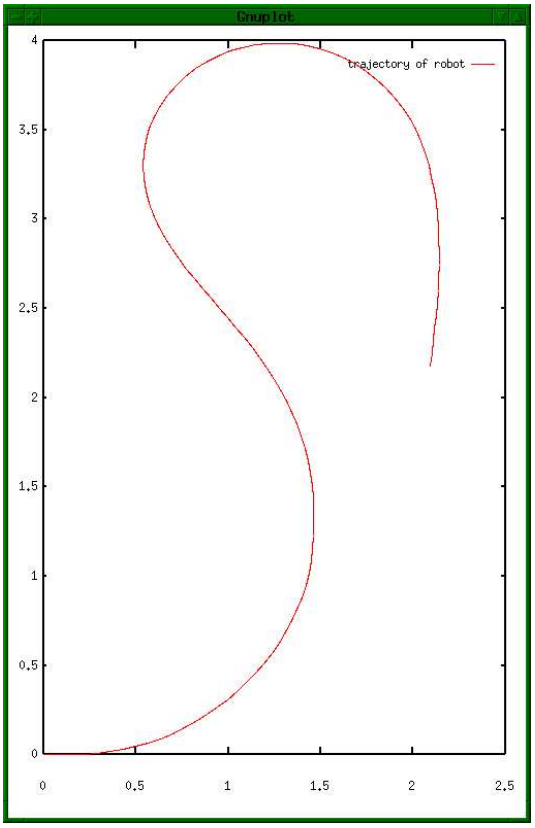


Figure 5.18: The trajectory of the robot for $\zeta(t) = \langle 0.0, 0.0, 0.0, 0.0, 0.0\rangle$ and $\zeta_{target} = \langle 2.0, 2.2, 270.0, 0.0, 0.0\rangle$

Learning We want to learn a policy-function, a direct mapping from the robot's current state and the robot's target state to the next command to be executed. We use multi layer artificial neural networks (see appendix A.2 for details) and the RPROP algorithm [Riedmiller and Braun 1993] because of the network's abilities in extrapolation. Considering the start state at $x = 0, y = 0$, and $\varphi = 0$ in a local system we can reduce the input dimension of the function to be approximated to 7 by converting the target states' x, y, φ into that local system (this means we regard $\Delta x, \Delta y, \Delta\varphi$). We can further reduce the input dimension to 5 by neglecting the rotational velocities. If we scale the network's output to a certain translational target velocity if the robot is close to its target position we can even reduce the input to three dimensions. This time, we have to use a more complex network with 3-6-8-6-1 topology for learning. The output of the network gives us the quotient $q = \frac{V_{rot}}{V_{tr}}$.

Exploitation During exploitation, we put $\Delta x, \Delta y$, and $\Delta\varphi$ into the trained neural network. The network then computes a quotient q that tells us how to set V_{rot} in relation to $V_{tr} = 0.75$ (meters per second). If the absolute value of V_{rot} is

more than 180 (degrees per second) we set $V_{rot} = 180$ and $V_{tr} = \frac{V_{rot}}{q}$. Otherwise we set $V_{tr} = 0.75$ and $V_{rot} = q \cdot V_{tr}$. Altogether this means we directly map the input to an action by performing policy-function approximation as proposed in section 4.4.6.

Figures 5.16 to 5.18 show trajectories obtained from the learned low-level-controller. While the trajectories in figures 5.16(a) and 5.17 show convincing results the trajectories to distant points in state space (figures 5.16(b) and 5.18) are obviously not optimal. This is caused by the fact that the training data covered local navigations only. This has two reasons:

(1) The purpose of the learned controller is to find a smooth trajectory from one state to another state that is relatively *close* (as mentioned at the beginning of this section). We want to employ this controller to navigate to states proposed by a higher-level path planning module.

(2) We used a huge amount of training data for our experiments. The time to train the neural network was around one week on a 500 MHz double Pentium machine. More training data means even more training time. In this case learning is not convenient any more.

The fact that the controller reaches distant target states too (without explicitly having learned these cases) demonstrates its robustness and its qualified abilities in extrapolation.

Robot Platform For the experiments in this section we used a Pioneer I robot [Pioneer 1998].

5.6 Multi Robot Path Planning for Static and Dynamic Environments

5.6.1 Introduction

Path planning is one of the fundamental computational problems in autonomous robot control. A wide variety of path planning algorithms have been developed and studied. Several textbooks, such as [Latombe 1991], give systematic accounts of the navigation problem and possible solution methods.

While the computational properties of planning algorithms, that is their correctness, their completeness, the optimality of their results, and their time and space complexity have been thoroughly investigated, the problem of how to choose the

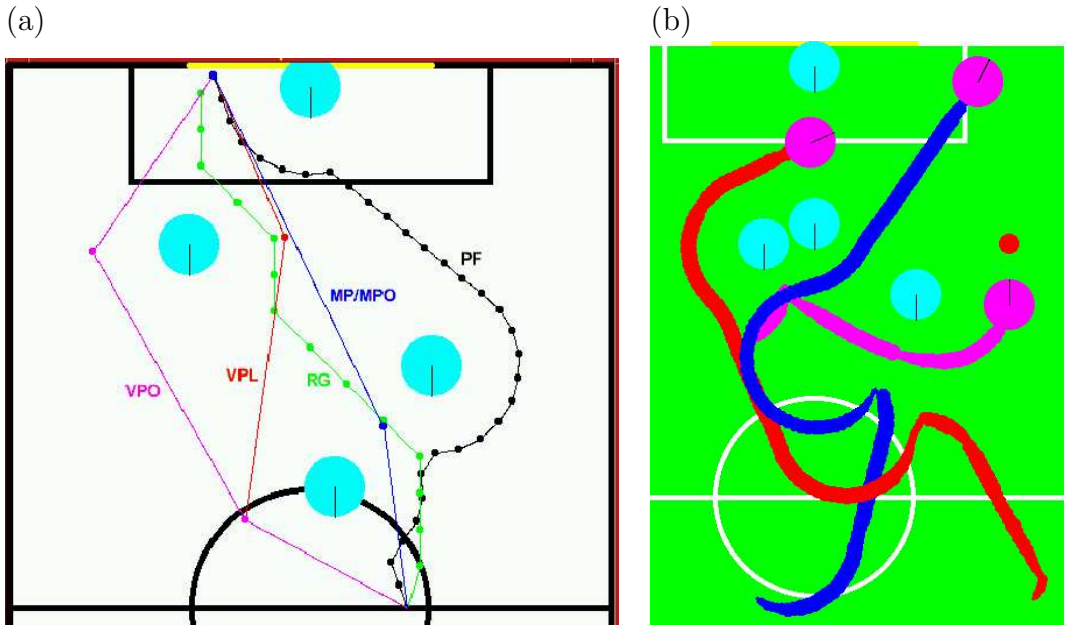


Figure 5.19: Robot navigation tasks in a typical robot soccer scenario: Subfigure (a) shows navigation plans for one robot proposed by different path planning methods. Subfigure (b) shows the trajectories that the robots followed in a multi robot scenario. The width of the trajectory indicates the robot's translational velocity.

right planning algorithm for a particular application and parameterize it optimally has received surprisingly little attention [Buck *et al.* 2001a].

In our work we regard path planning as a control problem. In fact, path planning and control have a lot in common: Both have start states and target states, both depend on a particular distance metric and on the (mostly not well known) dynamics of the machine at hand, and both have to generate a path through a multi-dimensional state space to reach a defined target state.

Consider, for example, navigation problems that arise in autonomous robot soccer (see appendix B for details on the RoboCup challenge). In robot soccer (mid-size league) two teams of four autonomous robots play against each other. In order to make a particular play, the robots of one team have to perform a joint navigation task, given by a target position for each robot. Thus to make competent plays the robots must be capable of solving the following category of navigation planning problems: Given the current positions of three robots, the field players, and their respective target positions, compute a navigation path for each robot such that when the three navigation plans are executed concurrently the expected time for reaching the goal configuration is minimal. Figure 5.19(a) depicts a single robot navigation task in a typical game situation and the navigation plans proposed by different navigation planning algorithms. The figure illustrates that the paths computed by the different methods are qualitatively very different. While one

path is longer and keeps larger distances to the next obstacles another one is shorter but requires more abrupt directional changes. The performance that the paths accomplish depends on many factors that the planning algorithms have not taken into account (see figure 5.19(b)). These factors include whether the robot is holonomic or not, the dynamical properties of the robot, the characteristics of changes in the environment, and so on.

The conclusion that we draw from this example is that the choice of problem-adequate navigation planning methods should be based on empirical investigations, that is exploration runs with different algorithms. In experience-based control, the choice of an algorithm corresponds to an action. In this section we develop a method for choosing the appropriate path planning algorithms (=actions) based on experiences gained from explorations.

Because in many multi robot applications the path planning tasks are heterogeneous and there are typical distributions of navigation tasks we develop a feature language that allows us to classify navigation tasks along dimensions that challenge planning methods. The features of a path planning task represent the current state. The state space includes all possible states. We show how a robot can make up classes of path planning tasks and choose the right planning mechanisms for a given task. Therefore we learn a policy-function (see section 4.4.6) that maps from the space of features of navigation tasks to an appropriate planning algorithm which is, in the context of our learning methods, an action.

5.6.2 Algorithms for Single Robot Path Planning

Let us now introduce the path planning methods that we will use in our subsequent experiments. We categorize these methods according to the objectives they try to optimize into four categories. We consider methods that are based on attraction-forces to the destination, ones that try to minimize path length, ones that consider a path as a sequence of circumnavigation steps, and ones that aim at maximizing the clearance of the paths.

Potential Fields

Potential fields are popular and often used means for path planning. The basic idea of potential field navigation planning [Khatib 1986, Hwang and Ahuja 1992] is to assign a value $\mathcal{V}(p)$ to each location p in the discretized environment. $\mathcal{V}(p)$ results from the superposition of an attractive force towards the destination and repulsive forces caused by obstacles:

$$\mathcal{V}(p) = b_{target} \cdot \delta_{target} + b_O \sum_i \frac{1}{\delta_{O_i}} \tag{5.35}$$

b_{target} and b_O are factors to weight the influence of the distances, δ_{target} (distance to the target) and δ_{O_i} (distance to obstacle O_i).

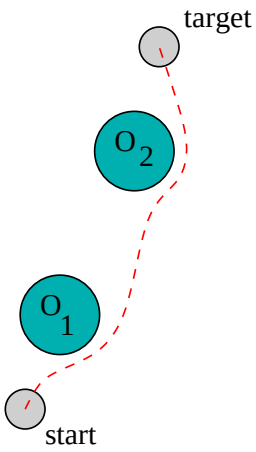


Figure 5.20: Potential field path planning.

The navigation plan then is the steepest descent path from the robot's current state to its target state (figure 5.20). A drawback of the basic algorithm is the likelihood of local minima which causes the algorithm to return paths that get stuck in a local optimum. These problems have been eliminated in a number of extensions of the basic algorithm (see for example [Tournassoud 1986, Volpe and Khosla 1990, Barraquand *et al.* 1992]). The plans generated by potential field methods trade off safety, the distance to the closest obstacles, and path length.

Backward Gradient Approach

Another class of planning algorithms aims at minimizing the path length [Lengyel *et al.* 1990, Konolige 2000].

These algorithms tessellate the environment into small grid cells and assign to each grid cell the length of the shortest path to the destination (figure 5.21). The target state is labeled with a value of zero. All other states of the grid are initially labeled with very high values. The algorithm starts with the target state and in each iteration visits all adjacent states. The value $\mathcal{V}(p_{i+1})$ for a state p_{i+1} adjacent to state p_i is infinity if p_{i+1} is inside an obstacle and

$$\mathcal{V}(p_{i+1}) = \min(\mathcal{V}(p_i) + 1, \mathcal{V}(p_{i+1})) \tag{5.36}$$

else.

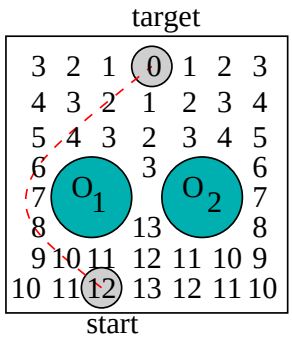


Figure 5.21: The shortest path method for path planning.

This class of planning methods has drawbacks in that the methods do not have a notion of path curvature. This may result in paths that require frequent changes of the robot's direction and speed. In addition, in their basic realization the methods prefer paths that are close to obstacles. This problem can be alleviated by either artificially growing the obstacles or by viewing navigation as a Markov decision process [Kaelbling *et al.* 1996] where the actions have nondeterministic effects. The latter approach, however, tends to yield computationally expensive solution methods. Further, the grid-based backward gradient approach causes

serious computational problems if applied to state spaces with more than three dimensions.

Circumnavigation of Obstacles

This class of methods considers path planning as finding a sequence of navigation actions that circumnavigate the obstacles that intersect the straight line paths to the destinations. The A* algorithm [Hart *et al.* 1968] computes the shortest path using the given points for circumnavigation.

This category of path planning methods includes the viapoint method [Schweikard 1992] (figure 5.22) and the elastic band algorithm (for dynamic obstacles) and visibility graph planning (for static obstacles) [Latombe 1991]. These methods sometimes cause problems when the start state or the target state is too close to an obstacle.

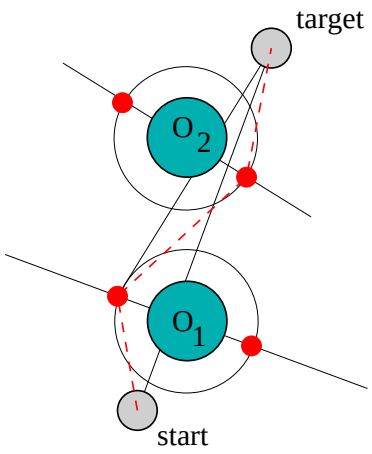


Figure 5.22: The viapoint algorithm for path planning.

Maximum Clearance Algorithms

This category of navigation planning algorithms aims at the computation of paths that keep maximal distance to the obstacles.

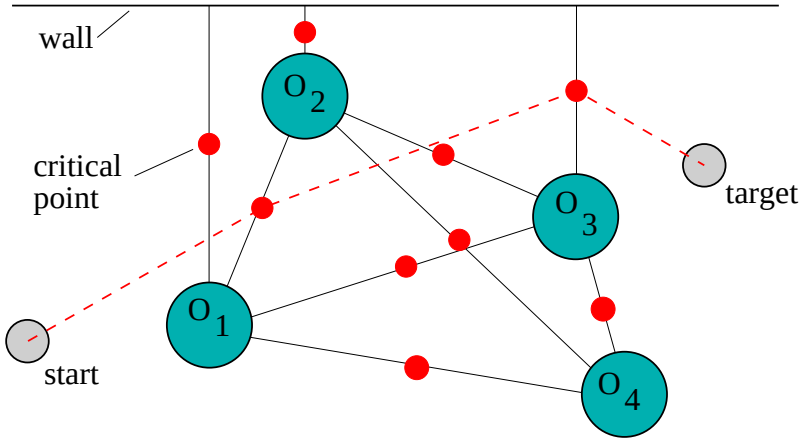


Figure 5.23: The maximum clearance algorithm.

A well known member of this family is the *Voronoi path planning* algorithm [Latombe 1991] that guides the robots on paths that have equal distances to

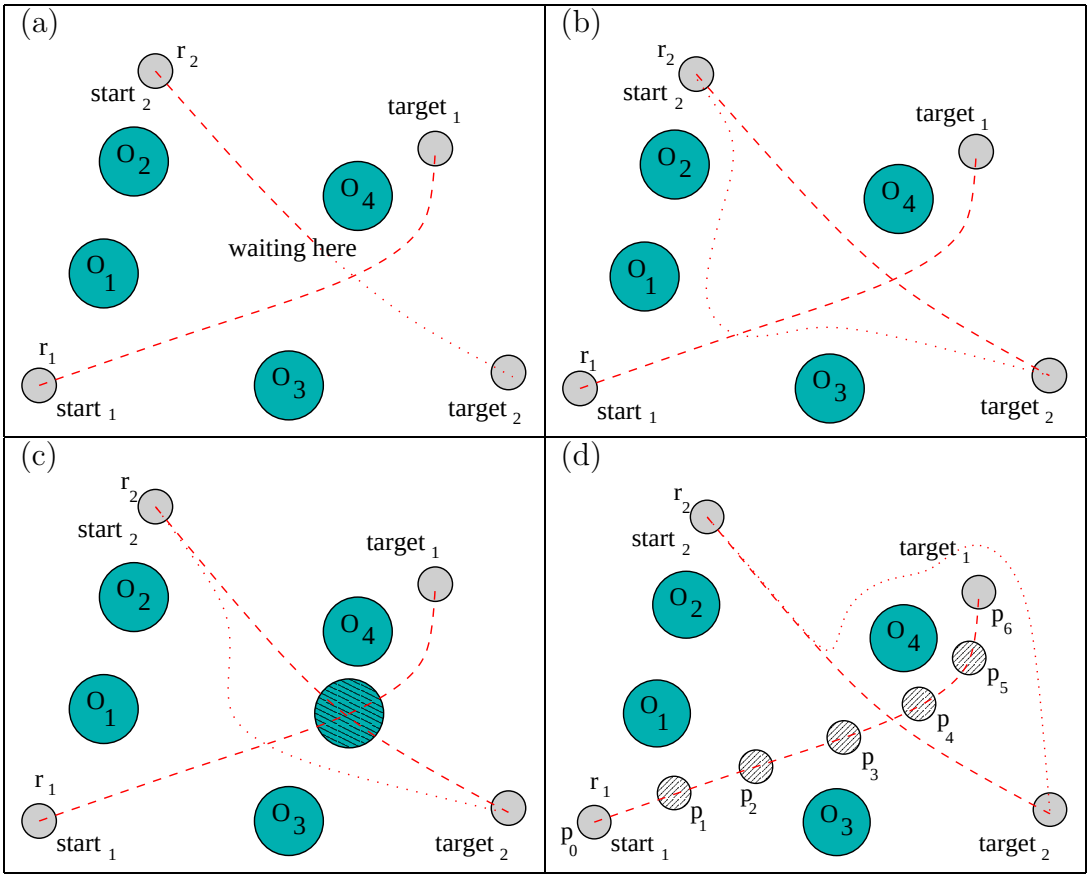


Figure 5.24: Several strategies for merging and repairing multi robot navigation plans by combining single robot navigation plans. Subfigure (a): Waiting strategy. Subfigure (b): Temporary targets. Subfigure (c): Hallucination of obstacles. Subfigure (d): Priority-based perception.

the closest obstacles. Another planning algorithm that aims at maximizing the clearance in the face of moving obstacles is the *maximum clearance* planning algorithm used in [Buck *et al.* 2000] that we use in our experiments as well. In this algorithm *critical points*, points that have equal distance to at least two obstacles, are computed and proposed as intermediate points on the way to the target (see figure 5.23). Due to the large average distance to obstacles these algorithms tend to propose paths that have a lower curvature than those proposed by other algorithms at the cost that the paths are becoming longer.

5.6.3 Algorithms for Plan Merging and Repair

After having detailed the algorithms for single robot path planning we will now address the question of how to combine the individual plans in order to obtain a good performance on the joint navigation tasks. Therefore we employ a number

of methods that merge different single robot paths in order to achieve a consistent set of plans that can be executed in parallel. The methods employed contain:

Waiting

Simply combining the paths computed by each robot without taking further precautions entails the danger that two robots might collide if their paths intersect. When two robots are to reach an intersection at about the same time the simplest fix is to let one robot wait until the other robot has crossed the intersection. In figure 5.24(a) robot r_2 waits until robot r_1 has passed and then continues driving (dotted path). One can make this strategy smarter by assigning priorities to the different robots, such that the robot with the more urgent task can go first. For instance, [Buckley 1989] assigns higher priority to a robot that can drive on a straight line. In this case it can keep a better dynamic state.

The remaining methods try to revise the individual plans such that no negative interferences will occur. Again we assign priorities to the robots according to the importance of their navigation tasks and ask the robots with lower priority to revise their plans to avoid conflicts with the paths of the higher priority robots. We have considered three different methods for path revision. The first one modifies the path by introducing additional intermediate target points. The second one hallucinates additional obstacles at the positions where collisions might occur. The third one simply considers the other robot at its respective position as a static obstacle.

Temporary Targets

To avoid possible collisions, we can constrain a path by specifying additional intermediate target points. A natural constraint to impose on the path of the lower priority robot is that it is to intersect the path of the other robot behind the robot. This can be easily accomplished by setting an additional intermediate target point behind the robot with the higher priority. In figure 5.24(b) robot r_2 moves to a point behind robot r_1 as long as their paths intersect. The disadvantage with respect to the waiting strategy is that the paths of side stepping robots become longer. The advantage with respect to the waiting strategy is that the side stepping robots can keep a better dynamic state, that is they do not have to stop.

Hallucination of Obstacles

Another alternative for path revision is the insertion of additional artificial obstacles at the problematic path intersections. In this solution the robot with

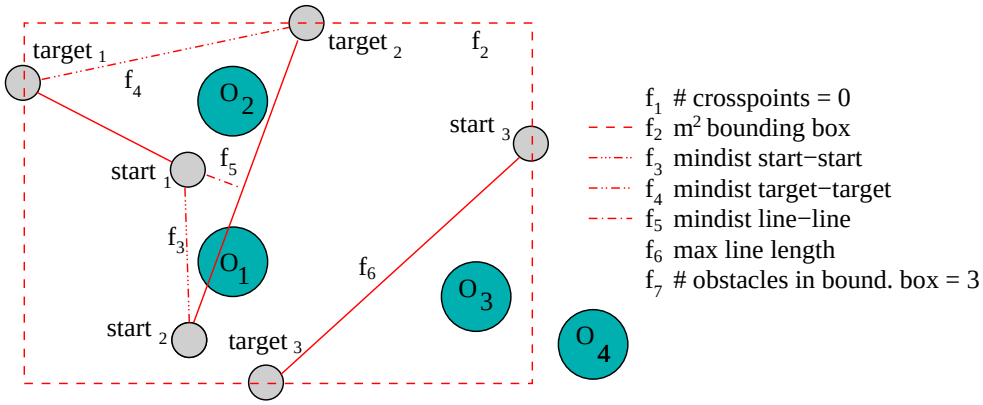


Figure 5.25: Visualization of navigation task features that are used for classifying navigation tasks. start and target points of the robots are indicated.

the lower priority hallucinates an obstacle at the intersection position and therefore plans a path around the critical area. The success of this path replanning technique depends on a competent placement of the intermediate target points. Figure 5.24(c) shows the insertion of an additional obstacle at the intersection point. However, new critical areas might arise from path replanning.

Priority-Based Perception

If we determine different priorities for the robots we let the robots with the lower priority perceive those with higher priority as obstacles. In figure 5.24(d) robot r_2 (low priority) repeatedly perceives robot r_1 (high priority) as an obstacle. The lower priority robot (r_2) has to replan its path. However, this insertion tactics has a drawback in that it might cause the lower priority robots to repeatedly replan or cause paths with unnecessarily high curvature.

5.6.4 A State Space for Multi Robot Path Planning Tasks

As we can see in the preceding subsection, a large variety of different single robot navigation planning and plan merging methods exists. All these methods have different strengths and weaknesses, and they make different assumptions about the navigation problems at hand. This observation suggests that we need a state space of characteristical features that describe the properties of navigation problems in a given robot control application and use these features to select the appropriate new planning method.

In our investigations we will use seven different features to describe a navigation task (see figure 5.25). These features are:

(f_1) The number of intersections between the line segments that represent the navigation task.

(f_2) The size of the bounding box of the navigation task.

(f_3) The minimal linear distance between the different start states of the robots ($\min_{i \neq j} \delta(start_i, start_j)$).

(f_4) The minimal linear distance between the different target states of the robots ($\min_{i \neq j} \delta(target_i, target_j)$).

(f_5) The minimal linear distance between the line segments that represent the navigation tasks. This is zero if lines intersect.

(f_6) The maximum length of the linear distances of the individual navigation tasks ($\max_i \delta(start_i, target_i)$).

(f_7) The number of obstacles in the bounding box of the joint navigation task.

The number of intersections between navigation tasks gives us a measure of the expected complications caused by negative interactions of the individual navigation tasks (1). The size of the bounding box, the minimal linear distances between different start states as well as target states is intended to provide us with a measure of crowdedness caused by the individual navigation tasks (2-4). Navigation tasks with line segments that are close to each other can be expected to require a higher degree of synchronization (5). The maximum length of the linear distances of the individual navigation tasks gives us a crude measure of the expected duration of the joint navigation task (6). Finally, the number of obstacles in the bounding box of the joint navigation task should be correlated with the necessary jinks (7).

5.6.5 Empirical Investigations in Robot Soccer

For our control task, the action space is given by

$$\mathcal{A} = \mathcal{A}_{srpm} \times \mathcal{A}_{prm} \tag{5.37}$$

An action consists of a single robot path planning method (*srpm*) and a plan repair method (*prm*). $\mathcal{A}_{srpm}$ contains the algorithms introduced in section 5.6.2 while $\mathcal{A}_{prm}$ contains the methods of section 5.6.3:

$$\mathcal{A}_{srpm} = \{\text{potential field}, \text{backw. gradient}, \text{circumnav.}, \text{max. clearance}\} \tag{5.38}$$

$$\mathcal{A}_{prm} = \{\text{waiting}, \text{temp. targets}, \text{hallucination}, \text{pr.based perception}\} \quad (5.39)$$

The components of the state space have been defined in section 5.6.4 by the features $f_1, \ldots, f_7$. Thus a state is given by

$$\zeta = \langle f_1, \ldots, f_7 \rangle \quad (5.40)$$

After having introduced different navigation strategies and a state space for characterizing multi robot navigation problems we will now try to assess the strengths and weaknesses of the different methods in a particular application: Autonomous robot soccer. We will do so by making a simple quantitative comparison with respect to the average performance of the individual methods. We try to find clusters of navigation tasks within the state space that the individual methods solve well or poorly. Finally, we use that learned characterization of the kind of navigation problems that a method solves well in order to choose the navigation strategies in problem specific ways. We will show that such a deliberating navigation planner outperforms the individual methods that it uses.

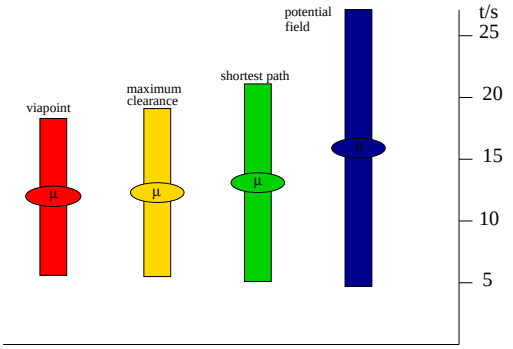


Figure 5.26: Mean values μ and standard deviation (both in seconds) of the compared path planning algorithms.

Exploration In our experiments we set the number of obstacles to $|\mathcal{O}| = 4$. This is because the team size in the RoboCup mid-size league is four. We carry out experiments with $|\mathcal{R}| = 3$ robots (resulting from 3 field players in RoboCup). All our experiments underlie the same randomly generated situations: The soccer field has a length of ten meters and a width of four meters. A robot starts at a randomly defined state in its configuration space and needs to move to an also randomly defined target state. The obstacles move linearly from one randomly generated start point to a randomly defined target. If an obstacle reaches its target a new target is defined immediately. Obstacles move with a randomly chosen but constant velocity from one point to another. We apply fix-action exploration (see section 4.3.3) by performing each navigation task with every single method (=action) and recording the time needed for completion. Figure 5.26 pictures the mean value and the standard deviation of the time resources required to complete a joint navigation task using the different single robot planning methods. The data for the statistics was acquired by performing 1000 randomly chosen navigation problems.

The results show that based on the empirical data we cannot determine a single method (a certain action) that outperforms the other ones in a statistically significant way. This suggests that we should try to identify specializations of the navigation problems (regions of the state space) for which one specific planning method (a certain action) outperforms the other ones. In the following we will look at this problem.

Learning A natural way for encoding a predictive model of the expected performance of different navigation planning methods in a given application domain is the specification of rules that have the following form:

if $c_1 \wedge \ldots \wedge c_n$
then fastest-method($\langle$*srpm,prm* $\rangle$)

In this rule pattern the c_i represent conditions over the features that we use to classify navigation problems (see section 5.6.4). The **then**-part of the rule asserts that for navigation problems that satisfy the conditions c_i the combination of the single robot planning method *srpm* together with the plan repair method *prm* can be expected to accomplish the navigation task faster then any other combination of single robot planning and plan repair method.

The advantage of a predictive model that consists of such rules is that it can be learned automatically by decision tree learning [Quinlan 1986] (see sections 5.8.3, A.5). For our experiments, we have used the public domain version of Quinlan's C4.5 algorithm. We obtain the data set that is necessary for the learning task in the following way.

(1) Generate a random navigation task from a given distribution of navigation tasks and compute the feature vector (state) $\zeta(t) = \langle f_1, \ldots, f_7 \rangle$ of the navigation task.

(2) Solve the navigation task with each combination $a = \langle srpm,prm \rangle$ of the single robot planning methods and plan repairs to be investigated.

(3) Store the data record $\langle \zeta(t), a(t) \rangle$ where $a(t)$ is the combination of single robot planning methods and plan repairs that achieved the best performance in the data set that is used for learning the decision tree.

Using this data collection method we have collected a training set of 1000 data records and used it for decision tree learning. From this training set the C4.5 algorithm with standard parameterization and subsequent rule extraction has learned a set of 10 rules including the following two:

1. **if** there is one intersection of the navigation problems
   $\wedge$ the navigation problems cover a small area ($\leq 10.7m^2$)
   $\wedge$ the target states are close to each other ($\leq 1.1m$)
   $\wedge$ the start/target state distances are small ($\leq 5m$)
   then fastest-method($\langle$*potential field,temp. targets*$\rangle$)

2. **if** there is no intersection of the navigation problems
   $\wedge$ the navigation problems cover a medium area
   $\wedge$ the distance between target states is moderate
   then fastest-method($\langle$*max. clearance,temp. targets*$\rangle$)

The first rule essentially says that the potential field method is appropriate if there is only one intersection and the joint navigation problem covers at most one fourth of the field, and the target states are close to each other. This is because the potential field algorithm tends to generate smooth paths even for cluttered neighborhoods. The second rule says that for typical size navigation problems where the navigation problems do not intersect, the maximum clearance method performs well. The rationale of the second rule is the maximal clearance causes the individual robots to take similar paths and therefore it becomes more likely that the robots pass the intersection area at the same time.

The accuracy of the ruleset for predicting the fastest navigation method is about 50% both for the training and the test set. A substantially slower algorithm (more than 110% of the optimal time) was chosen in less than 10% of the cases. The inaccuracies of the rules have several reasons. First, the state space as it has been introduced may not be expressive enough. We can expect that the accuracy of the rules can substantially be increased by adding additional features to the state space. Second, in many navigation problems different methods achieved almost the same performance (one might suppose this a priori by having a look at the bars of figure 5.26). In those cases we only selected the best one even when the margins where very narrow. Obviously, these data records are very noise sensitive. Third, in many runs collisions were caused by dynamic obstacles and have caused robots to get stuck. These runs resulted in outlier results that are not caused by the planning methods. Thus, for higher accuracy those runs have to be handled differently.

The conclusion that we draw from this experiment are that even with a crude feature language, without sophisticated data transformations and outlier handling a robot can learn useful predictive models for the performance of different navigation methods in a given application domain.

Exploitation An obvious idea for exploiting the predictive model that we have learned is the implementation of a hybrid navigation planner that uses a set of navigation methods and picks for every single navigation problem the navigation methods (=action) that are proposed by the decision tree. The decision tree then

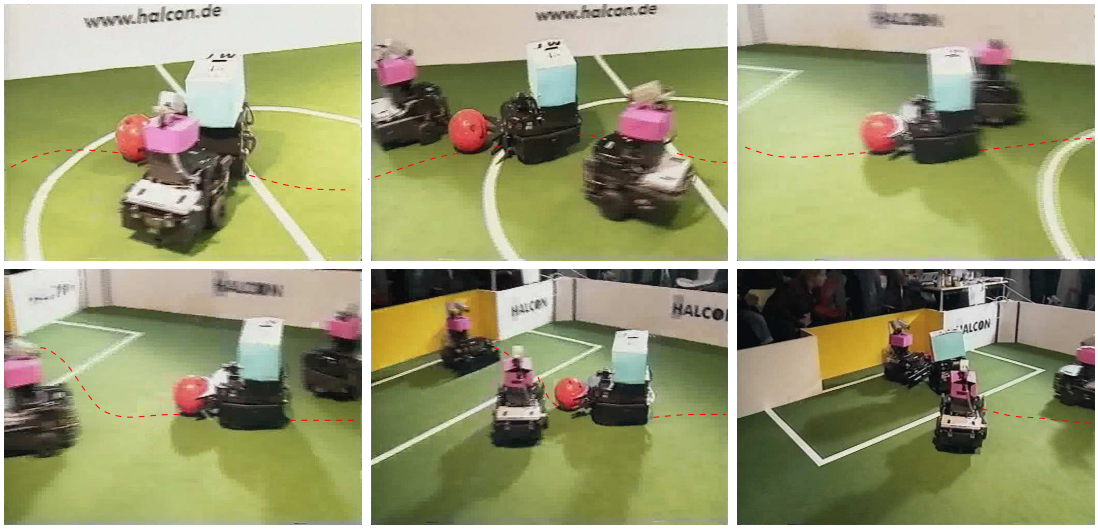


Figure 5.27: Real robot path planning under competitive conditions. The pictures are from a friendly game with the CS Freiburg team [Weigel *et al.* 2002] in September 2001. The video sequence can be viewed at *www9.in.tum.de/archive/agilo/RoboCup2001_20.mpg*.

represents a direct approximation of a policy-function (see section 4.4.6). Thus for the next experiment we have implemented such a hybrid navigation planner and compared its performance with the performance obtained by the individual navigation methods. Figure 5.27 shows one of our Pioneer I robots planning a path under competitive conditions.

We have performed a bootstrapping t-test based on 1000 different joint navigation tasks in order to empirically validate that the hybrid navigation planner performs better than the individual planning methods that we are using. Based on these experiments we obtained a 99.9% confidence in the test set (99.9% in

Algorithm	Mean time values of 1000 training and 1000 test problems			
	TRAIN		TEST	
	μ/sec	significance level $P(\mu_{tree} < \mu)$	μ/sec	significance level $P(\mu_{tree} < \mu)$
Simple Potential Field	15.49	99.99 %	15.92	99.99 %
Shortest Path	13.36	99.99 %	13.14	99.99 %
Maximum Clearance	12.35	99.71 %	12.31	99.84 %
Viapoint	12.14	94.62 %	11.95	96.25 %
Decision Tree	11.64		11.44	

Table 5.2: Results of four evaluated algorithms and the trained decision tree. The significance level is based on a t-test.

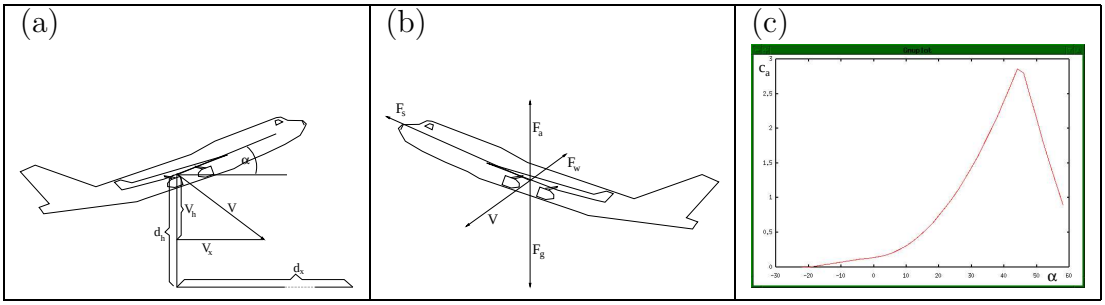


Figure 5.28: Subfigure (a): The state of the autolanding-task. Subfigure (b): Forces used for the simulation of the aircraft. Subfigure (c): The buoyancy-factor c_a as a function of α.

the training set) that the hybrid method outperforms the potential field method (with its respective parameterization). The respective probabilities for the backward gradient method are 99.9% (99.9%), for the maximum clearance method 99.84% (99.71%), and for the circumnavigation method 96.25% (94.62%). This means that our hypothesis that the hybrid planner dominates the other planning methods could be validated with statistical significance ($\geq 95\%$).

Robot Platform The robot platform used for the path planning tasks were Pioneer I robots [Pioneer 1998]. The training data was acquired by using the neural simulation described in section 3.5.1 and [Buck *et al.* 2002c].

5.7 The Aircraft-Autolanding-Task

After having applied experience-based learning techniques to a number of problems in the field of robotics, we now want to show that the proposed methods work well in other machine control domains too. For instance, the landing of an aircraft is a very difficult control problem for the following reasons: If an aircraft is landing its velocity both in horizontal and in vertical direction must be very small. But to get a small sinking-velocity a high horizontal velocity is necessary. Further the landing position and angle must satisfy some very special constraints. Some airports have only very short landing strips. Thus landing must occur in a limited space. For all these reasons, looking ahead is inevitable in this control task. Moreover, the target state can definitely not be reached by any random or simple policy. Even a pilot must take a long and extensive training to satisfy the requirements for safely landing an aircraft.

In the autolanding-task [Buck *et al.* 2002a] we strongly simplify the landing of a real aircraft by regarding an only five-dimensional state space in our simulations. Nevertheless, our simulation is quite realistic concerning the basic problems of

Variable	unit	range ζ_{start}	range ζ_{target}
d_x	km	$\in [20, 30]$	$\in [0, 3]$
d_h	km	$\in [6, 8]$	$= 0$
α	degrees	$\in [-10, +10]$	$\in [0, +30]$
V_x	km/h	$\in [500, 800]$	$\in [0, 300]$
V_h	km/h	$\in [-200, +100]$	$\in [-25, 0]$

Table 5.3: Constraints for start and target states.

controlling an aircraft. A state includes the distance to the end of the landing strip projected on the ground (d_x), the height of the aircraft (d_h), the angle of the aircraft ($\alpha = 0$ means horizontal flight), the horizontal velocity V_x, and the vertical velocity V_h (see figure 5.28(a)):

$$\zeta = \langle d_x, d_h, \alpha, V_x, V_h \rangle \tag{5.41}$$

Even with this simplification it is extremely difficult for a human to safely land an aircraft (by means of the keyboard of a PC) simulated by the means described below (see paragraph *Analytic Simulation*, equations (5.44) to (5.48)). In contrast to the work described in [Perez-Uribe 1997] our task is not to follow a given trajectory but to *find* a trajectory *and* follow it towards the target state.

The action space is two-dimensional and consists of the power *pow* that controls the propulsive force and the rotational factor *rot* that can make the aircraft rise or sink by adjusting α:

$$a = \langle pow, rot \rangle \tag{5.42}$$

pow must be out of $[0, F_{s_{max}}]$ and $rot \in [-10, +10]$, where $F_{s_{max}}$ is the maximal propulsive force of the aircraft (see table 5.4).

The scenario is the following: An aircraft is placed at ζ_{start}. Then, the control policy must land the aircraft at ζ_{target}. Table 5.3 specifies the constraints for the start state and the target state.

Exploration During forward exploration a simple initial policy never managed to reach a target state. Trials of humans controlling of the simulated aircraft via the keyboard of a PC remained unsuccessful to. These fact forces us to use backward exploration (section 4.3.2). Employing this method trajectories backwards from the target are obtained. Only one of the exploration runs was used to generate patterns for learning. Patterns are generated in order to approximate a value-function. We assign values to states according to

$$\mathcal{V}(\zeta(t)) = \begin{cases} +1 \;\; if \;\; \zeta(t) \in \mathcal{S}_{target} \\ \gamma \cdot \mathcal{V}(\zeta(t+1)) \;\; else \end{cases} \tag{5.43}$$

Learning The patterns obtained from the successful trajectory (consisting of state and corresponding value) are used in order to train a network of radial basis functions (see appendix A.3 and section 5.8.2 for details). This time, we employ radial basis functions for the approximation of the value-function because they work as attractors for given trajectories. Further they tend to return low values for states not visited before and do not exaggerate. The network's topology is optimized automatically and consists of a maximum of 80 Gaussian kernels.

Exploitation Actions are chosen according to equation (4.9) during exploitation. The trained value-function $\mathcal{V}$ is partly depicted in figure 5.29. In this figure a typical trajectory of a landing aircraft can be viewed. Figure 5.30 shows an example for the development of the aircraft's state over time. In more than 500 runs the value-function always attracted the aircraft and lead it towards the target.

An animation related to the above experiment can be found at
http://www9.in.tum.de/archive/agilo/AircraftAutolanding.gif.

Analytic Simulation

The simulator used for the above experiments relies only on the basic physical rules concerning forces and acceleration. It cannot be compared to a real flight simulator. Nevertheless the behavior of the aircraft, to a surprisingly high extent, is realistic.

This analytic simulation (see section 3.3.1) is based on the equations (3.2) and (3.3). The following forces are taken into account (see figure 5.28(b)).

Gravitation The force of gravity is computed by $F_g = mg$, where m is the mass of the aircraft and g is the earth constant.

Propulsive Force The propulsive force F_s is directly controlled by the policy ($F_s = pow$). It ranges from 0 to $F_{s_{max}}$.

Buoyancy Buoyancy is computed by

$$F_a = \frac{1}{2}\rho \cdot c_a(\alpha) A_F \cdot |V|^2 \tag{5.44}$$

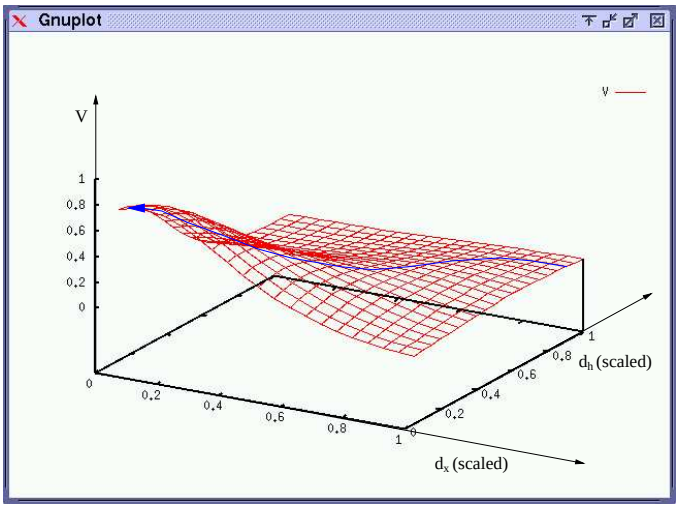


Figure 5.29: The approximated value-function $\mathcal{V}$ represented by radial basis functions. The function is depicted dependent on 2 dimensions only. d_x and d_h are scaled to $[0, 1]$ both.

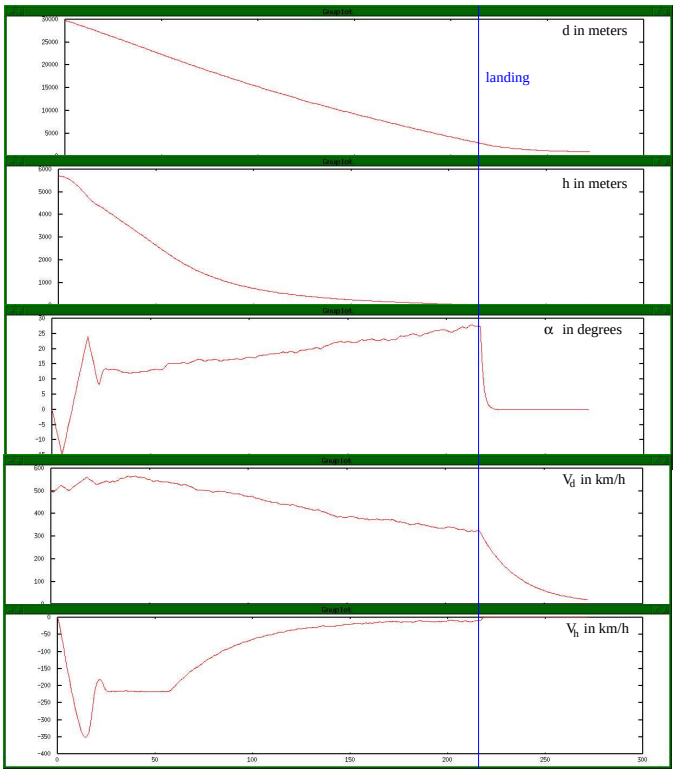


Figure 5.30: The state of a simulated aircraft over time. The vertical line marks the landing.

where ρ is the atmospheric pressure, $c_a(\alpha)$ is the buoyancy-factor (see figure 5.28(c)), A_F is the vane area of the aircraft, and V is the velocity of the aircraft.

Air Drag The force of air drag is computed by

$$F_w = -\frac{1}{2}\rho \cdot c_w A \cdot |V|^2 \tag{5.45}$$

where ρ is the atmospheric pressure, c_w is the air drag coefficient of the aircraft, A is the cross section of the vanes of the aircraft, and V is the velocity of the aircraft.

Acceleration The velocity of the aircraft is computed by

$$V(t + \Delta t) = V(t) + \frac{(F_g + F_s + F_a + F_w) \cdot \Delta t}{m} \tag{5.46}$$

where Δt is set to one second. Thus the acceleration directly depends on the forces.

Changes in State The change in state depends on the velocities and is computed as follows:

$$\langle d_x, d_h \rangle(t + \Delta t) = \langle d_x, d_h \rangle(t) + V(t + \Delta t) \cdot \Delta t \tag{5.47}$$

Once again Δt is set to one second. The angle of the aircraft (α) directly depends on the surface control devices which are controlled by the policy:

$$\alpha(t + \Delta t) = \alpha(t) + rot(t) \tag{5.48}$$

Table of Used Constants For our simulation certain constants are used. Each constant's value and meaning is described in table 5.4. Most values correspond to the values of a filled real Boeing 747.

The software for the simulation is accessible at *http://www9.in.tum.de/archive/agilo/AircraftAutolandingSimulation.tar* and may be used for research purpose only and without any support or warranty. It is written in C++ and runs at least under Linux, Solaris, and HP-UX.

Var.	Meaning	Value	Unit
ρ	atmospheric pressure	1.3	kg/m^3
c_w	air drag coefficient	2.0	-
c_a	buoyancy constant	$f(\alpha)$	-
g	earth constant	9.81	m/sec^2
A	cross section of vanes	10.0	m^2
A_F	vane area of a 747	528.20	m^2
m	mass of a filled 747	350000	kg
$F_{s_{max}}$	max. propulsive force ($4 \times 257.6\ kN$) (engine GE CF6-80C2B1F)	1030.4	kN

Table 5.4: Constants used for the simulation of the autolanding-task.

5.8 Software for Experience-Based Control

In the following we shortly describe the software used for the applications above.

5.8.1 n++ (Multi Layer Perceptrons)

The n++ neural network simulator [Riedmiller 1995, Riedmiller 1999] is a non-commercial C++ library for neural function approximation. It was developed at the University of Karlsruhe in the mid-90s and has successfully been used by researchers and students of various affiliations. In the following we describe the special features of n++. For a general introduction to neural networks see appendix A.2.

The basic features of n++ include the construction of the topology of a network, the computation of the output of a network, error backpropagation, automatic adaptation of the networks' weights, and loading and saving of networks. The software can be included in any C++ application or can be used separately. Special features include

- Simulation of multiple neural networks in parallel.
- Different types of initializations for weights and connections.
- RPROP algorithm.
- Momentum.
- Weight decay.

- Learning with validation data (for early stopping).
- Scaling of input and output neurons.
- Support for temporal difference learning.
- Automatic generation of gnuplot files containing the error.
- Learning with multiple runs.

5.8.2 RBF++ (Networks of Radial Basis Functions)

RBF++ [Buck 2002] is a noncommercial tool for function approximation by means of radial basis functions.
It can be obtained from *http://www9.in.tum.de/archive/agilo/RBF++_1.2.tgz*.
For a general description of networks of radial basis functions see appendix A.3.

The basic features of RBF++ include the construction of the topology of a network, the computation of the output of a network, error backpropagation, automatic adaptation of the networks' weights, and loading and saving of networks. Special features of RBF++ include

- Simulation of multiple RBF-networks in parallel.
- RPROP algorithm.
- Learning with validation data (for early stopping).
- Automatic merging of Gaussian kernels.
- Automatic insertion of new Gaussian kernels to reduce the error to a specified level.
- Automatic deletion of irrelevant Gaussian kernels.
- Smart initial positioning of Gaussian kernels on characteristic patterns.

5.8.3 C4.5 (Decision Trees)

The C4.5 decision tree library [Quinlan 1986] implements decision trees for classification tasks. A public domain version for download exists in the internet: *http://www.cse.unsw.edu.au/~quinlan/*. For a general description of decision trees see appendix A.5.

C4.5 generates rules for classification using a set of training data. The program is given files defining classes, features and examples. Special features include

- Generation of decision trees in batch mode or iteratively.
- Automatic pruning of trees to simplify them.
- Evaluation of generated trees with validation data.
- Construction of rules from decision trees generated.
- Automatic optimization of rules.
- Evaluation of generated rules with validation data.

Chapter 6

Layered Experience-Based Control

6.1 Introduction

In the previous section we have seen a number of problems and respective implementations of solutions based on experience-based control. In subsection 2.2.5 we already referred to the problem of highly complex tasks briefly. In this section, we explain how to apply experience-based learning techniques successfully to complex problems that cannot be learned in one shot.

A standard approach to reduce the complexity of a learning task is to divide it into different layers. Then, in each single layer, problems of lower complexity have to be solved. This method is widely reported in the literature [Arkin 1989, Arkin 1998, Dudek and Jenkin 2000, Murphy 2000] and has been used in many applications [Stone 2000, Riedmiller and Merke 2002, Beetz *et al.* 2002, Burkhard *et al.* 2002]. For example, if we want robots to learn to play soccer, it makes no sense to let the robots autonomously learn the whole task by only providing the information whether a goal was scored or not. It is much more better to let the robots learn which high level action (dribble shoot to the goal,...) to perform in which situation after learning which low level actions to perform in order to make the single high level actions work most efficiently. As we can see in this example, the different layers, namely the high-level actions and the low-level actions, are not completely independent. The performance of every single high-level action depends on the reliability of the policy learned in the lower layer. If the robot has learned how to dribble but not how to shoot the ball it makes no sense to try to shoot even if the robot is in a good position for a shot.

In the context of this work we regard layered learning not only in terms of one learning agent but in applications where at least two learning agents cooperate in

order to solve a common task. In the latter case layered learning means to divide a complex learning task into several tasks that can be performed by different agents in parallel [Buck *et al.* 2002d].

6.2 Gain and Feasibility

If we perform layered experience-based learning we have to consider the different state spaces $\mathcal{S}^l$ and action spaces $\mathcal{A}^l$ of the respective layers l ($l \in \{I, II, ..\}$).

Gain The term *gain* defines the appropriateness of an action a^l in state ζ^l in a certain layer l considering the respective layer *only*. It describes how good it would be to perform the action a^l no matter what amount it takes to perform it and what the predicted rate of success might be. In case of a soccer robot the action shoot2goal always has a higher gain than for instance dribble because a goal has a greater impact on the result of the game. Here, the time it takes to execute shoot2goal is not considered. Represented by a Q-function the gain is given by $Q^l(\zeta^l, a^l)$.

Feasibility The *feasibility* defines the appropriateness of an action a^l in state ζ^l in a certain layer l considering the state $\zeta^{l-1}(a^l)$ in the layer below resulting from the action a^l in layer l. The feasibility gives us a measure if it is feasible or what amount it takes to choose the action a^l. Represented by a Q-function the feasibility is given by $Q^{l-1}(\zeta^{l-1}(a^l), \pi^{l-1*}(\zeta^{l-1}(a^l)))$. Herein π^{l*} is the optimal policy for a particular layer l:

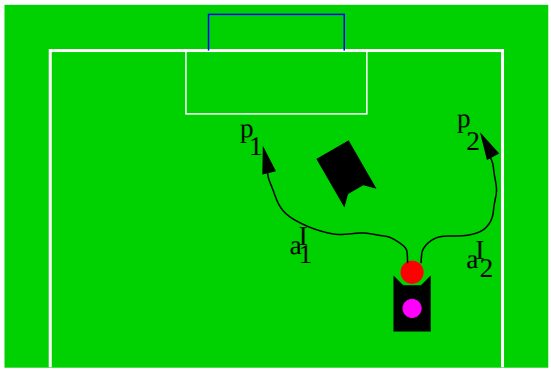


Figure 6.1: The robot possessing the ball can choose from two actions. The actions differ in gain and feasibility.

$$\pi^{l*}(\zeta^l) = \arg\max_{a^l \in \mathcal{A}^l} Q^l(\zeta^l, a^l) \qquad (6.1)$$

To illustrate the definitions of gain and feasibility we provide an example. Let us assume a soccer robot in possession of the ball as depicted in figure 6.1. The robot can choose from two actions in layer I:

$$\mathcal{A}^I = \{a_1^I, a_2^I\} \qquad (6.2)$$

Action a_1^I means to go to position p_1 while action a_2^I means to go to position p_2. In layer $I\!I$ the robot can specify its target velocity in meters per second and degrees per second. The action space is given by

$$\mathcal{A}^{I\!I} = [-1.0, +1.0] \times [-180, +180] \tag{6.3}$$

Obviously, position p_1 is much more better to score a goal than position p_2. This means a_1^I has a higher gain than a_2^I. On the other hand, the defender in figure 6.1 is more likely to hinder the player possessing the ball if it moves to p_1. As a consequence, the feasibility of the action a_1^I is lower than the feasibility of a_2^I. For instance let us assume the following values for gain and feasibility of the different actions.

	goto position p_1 (action a_1^I)	goto position p_2 (action a_2^I)
gain	$Q^I(\zeta^I, a_1^I) = 0.5$	$Q^I(\zeta^I, a_2^I) = 0.3$
feasibility	$Q^{I\!I}(\zeta^{I\!I}(a_1^I), \pi^{I\!I *}(\zeta^{I\!I}(a_1^I))) = 0.4$	$Q^{I\!I}(\zeta^{I\!I}(a_2^I), \pi^{I\!I *}(\zeta^{I\!I}(a_2^I))) = 0.8$

(6.4)

The decision of the soccer robot depends on how the different values of feasibility and gain are weighted. In the following we propose different methods to support decision making if there is more than one layer.

6.2.1 Combinations of Q-Functions

In order to be able to choose actions consistently we need to compute *one* global value out of the Q-functions of all layers. Dependent on the application the Q-functions might be combined by a sum or by a product. In the case of a weighted sum we maximize the following term:

$$\max_{a_i^I \in \mathcal{A}^I} \hat{Q}(\zeta^I, a_i^I) = \max_{a_i^I \in \mathcal{A}^I} [w^I \cdot Q^I(\zeta^I, a_i^I) + \sum_{l>I} w^l \cdot Q^l(\zeta^l(a^{l-I}), \pi^{l*}(\zeta^l(a^{l-I})))] \tag{6.5}$$

Herein the w^l are the weights of the Q-functions of the respective layers. $\hat{Q}$ evaluates action a_i^l and state ζ^I depending on the weighted sum of the Q-functions of all layers. For a system containing two layers the overall policy π^I in the higher layer is defined by

$$\begin{aligned} \pi^I(\zeta^I) = &\ \arg\max_{a_i^I \in \mathcal{A}^I} \hat{Q}(\zeta^I, a_i^I) \\ = &\ \arg\max_{a_i^I \in \mathcal{A}^I} w^I \cdot Q^I(\zeta^I, a_i^I) + w^{I\!I} \cdot Q^{I\!I}(\zeta^{I\!I}(a^I), \pi^{I\!I *}(\zeta^{I\!I}(a^I))) \end{aligned} \tag{6.6}$$

Along the same lines, we can use a product instead of a sum. In this case we maximize the term

$$\max_{a_i^I \in \mathcal{A}^I} \hat{Q}(\zeta^I, a_i^I) = \max_{a_i^I \in \mathcal{A}^I} [Q^I(\zeta^I, a_i^I) \cdot \prod_{l>I} Q^l(\zeta^l(a^{l-I}), \pi^{l*}(\zeta^l(a^{l-I})))] \tag{6.7}$$

Consequently, for a system containing two layers the overall policy π^I in the higher layer is given by

$$\begin{array}{rcl} \pi^I(\zeta^I) = & \arg\max_{a_i^I \in \mathcal{A}^I} \hat{Q}(\zeta^I, a_i^I) \\ = & \arg\max_{a_i^I \in \mathcal{A}^I} Q^I(\zeta^I, a_i^I) \cdot Q^{II}(\zeta^{II}(a^I), \pi^{II*}(\zeta^{II}(a^I))) \end{array} \tag{6.8}$$

The advantage of using a weighted sum is that each single layer can be parameterized depending on its importance. This is not possible if a product is used. On the other hand, using a product ensures that the Q-function of each layer has a certain value. If using a sum the Q-function of one layer can be zero for the best action overall. Usually, this is not desired.

6.2.2 Thresholds

If we want to ensure that the feasibility of an action is above a certain level and the best action fulfilling this requirement is chosen we can use thresholds to specify the minimum level of feasibility [Buck and Riedmiller 2000]. Then the policy π^I in layer I is given by

$$\begin{array}{rcl} \pi^I(\zeta^I) = & \arg\max_{a_i^I \in \mathcal{A}^I} \hat{Q}(\zeta^I, a_i^I) \\ = & \arg\max_{a_i^I \in \mathcal{A}_\Theta^I} Q^I(\zeta^I, a_i^I) \end{array} \tag{6.9}$$

where $\mathcal{A}_\Theta^I$ is the set of actions that exceeds the minimum level of feasibility:

$$\mathcal{A}_\Theta^I = \{a_j^I \in \mathcal{A}^I | Q^{II}(\zeta^{II}(a_j^I), \pi^{II*}(\zeta^{II}(a_j^I))) > \Theta^{II}\} \tag{6.10}$$

The advantage of this method is that a certain value can be guaranteed for the Q-function. However, for the special case of $\mathcal{A}_\Theta^I = \emptyset$ an exception-handling must be implemented.

6.3 Multi Agent Layered EBC

In the preceding subsection we provided several methods to combine the Q-functions of different layers in order to obtain a single value that we can use for choosing an action. In the following, we regard the scenario of layered

learning with multiple agents. The main advantages of multi agent systems over single agent systems are speed-up and fault tolerance [Cao *et al.* 1997, Dudek *et al.* 1996]. But without a reasonable coordination a multi agent system can even be less efficient than a single agent system (e.g. two robots block each other). Therefore [Mataric 1998a, Mataric 1998b] suggest that control in multi agent systems must be addressed as a separate, novel, and unified problem, not an additional 'module' within a single agent approach.

To underline the problems occurring in multi agent systems we take a look at an example first. Let us once again consider robot soccer. In contrast to the last example, we have a team of two robots now. Further we have one defender of an opponent team and the situation depicted in figure 6.2.

This time, the possible actions for robot 1 and robot 2 are go2ball or block_ball. The robots have to coordinate their actions such that one robot tries to get the ball (go2ball) and the second robot tries to hinder the opponent defender from attacking (block_ball) [1]. While robot 2 is closer to the ball robot 1 is in a better position to score a goal once it gets the ball. Here, we have to think about the coordination of the robots. In the depicted situation it makes no sense if both robots choose the action go2ball.

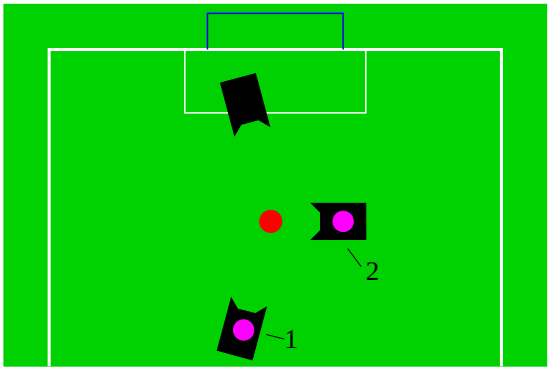


Figure 6.2: Robot 1 and robot 2 have to coordinate their behavior.

As a precondition for coordinated behavior we need to synchronize the perceptions of the single agents in such a multi-agent learning scenario. In terms of our example we have to ensure that the robots choose their actions based on shared information in order to act consistently.

6.3.1 Shared Information

Information acquisition of agents in general occurs by sensors or communication. To achieve consistency and cooperation in multi agent behavior some kind of common basis for situation assessment is necessary (e.g. synchronization or a shared world model) [Tews and Wyeth 2000]. [Young *et al.* 2002] distinguish between leader-following and behavioral schemes.

[1] In our example we consider only this possibility. Of course there are a lot more options.

Leader-Following Schemes

In the leader-following approach one agent organizes the whole coordination and assigns actions to all other agents of the team. Success strongly depends on the leader-agent and therefore fault tolerance is poor. If the leader-agent is out of order the whole system may collapse. [Chen and Luh 1994] provide an example for a leader-robot system.

Behavioral Schemes

In contrast to leader-following schemes, in behavioral schemes, agents autonomously decide what to do supported by more or less communication. This requires intelligent software and computational resources for *all* agents. Behavioral systems can be categorized in those with a shared model of the environment, those with a shared abstract description of the environment's state (e.g. at the planning level), and those without shared information. Examples for systems using global maps are the collaborative exploration systems described in [Burgard *et al.* 2000, Simmons *et al.* 2000]. [Alur *et al.* 1999] periodically exchanges information at discrete time intervals. Many other systems rely on communication too [Levesque *et al.* 1990, Jennings 1995, Saffiotti *et al.* 2000]. The system MAPS [Tews and Wyeth 1999] uses globally shared information remodeled in an abstract virtual space. Multi agent systems with a shared model of the environment require the computation of such a model. This is not trivial. In systems with a synchronization on an abstract description level we are forced to change the implementation everytime anything changes on the abstract level (e.g. priorities in planning). Furthermore it is argued that communication may lead to delays in information acquisition and can increase complexity and degrade multi robot systems [Kube and Zhang 1994, Sen *et al.* 1994, Garland and Alterman 1996]. However, using no shared information means to be dependent on an accurate state estimation of the environment.

Exchange of Key Features for a Consistent Assignment of Actions To meet the requirements for a consistent assignment of actions we rely on behavioral schemes using information shared among the agents of the team. Each agent receives its own sensor data and key features from its team mates via communication. Out of this information a model of the agent's environment is built. Thereby principally local information is used while the other agents' key features (if available) are used for evidence. Additionally the agent constructs a model of the environment from each other agent's local view by primarily relying on the key features of the respective agent. If those features are not available only local information is used and the system will keep working despite communication capabilities being corrupted.

If each agent has a model of the environment from each other agent's local view it can put itself in its team mate's situation. This mechanism allows an agent to consider its team mates' intentions in the process of choosing an action[2]. Moreover, if the same software is running on all agents each agent can compute its team mates' actions.

6.3.2 Exhaustive Search

In the following we consider a multi robot system (or multi agent system) $\mathcal{R}$. Herein $r_i \in \mathcal{R}$ means a particular robot *with its respective state*. Action selection from a set of actions $\mathcal{A}$ in the context of the multi robot system $\mathcal{R}$ means to find a mapping

$$\pi : \mathcal{R} \rightarrow \mathcal{A} \tag{6.11}$$

that assigns each robot (and its respective state) of the system a different action $a_j \in A$ in a way that the defined common goal can be performed with a maximal Q^{Π}-value over all robots. This means we regard the feasibility of an action depending on the robot performing it. $Q^{\Pi}(r_i, a_j)$ is the Q^{Π}-value (feasibility) for robot r_i performing action a_j. Hence we want to maximize the following term:

$$max \sum_{i=1}^{|\mathcal{R}|} Q^{\Pi}(r_i, \pi(r_i)) \qquad i \neq j \Leftrightarrow \pi(r_i) \neq \pi(r_j) \tag{6.12}$$

Since this combinatorial problem is NP-hard it is advisable to employ heuristics for maximizing the feasibility function. Assuming there is only a small number of robots (e.g. $|\mathcal{R}| = 4$) and a small number of actions this problem can be solved exactly by an exhaustive search over all combinations possible.

6.3.3 Priorities of Actions

In the case of an exhaustive search, we regard only the feasibility of an action depending on the robot, that is, we consider if the robot can execute the action and for what cost. Here we make the assumption that the performance of all actions is equally important. Obviously, in many real world applications this is not the case: In autonomous mine sweeping for example, disabling the mines close to a village or even a building is more important than disabling distant mines. In robot soccer, shooting the ball is prior to any other action in an offensive situation. Consequently, we construct a priority list of actions $\mathcal{A}^I_{pr}$ to support

[2]Interestingly, this ability distinguishes humans and apes on one side from all other mammals on the other side

the computation of π. $\mathcal{A}^I_{pr}$ contains all actions of $\mathcal{A}^I$ in the order of their priority. Below an example for $\mathcal{A}^I_{pr}$ is given.

action $\in \mathcal{A}^I$	Q^I(action)
a^I_1	1.0
a^I_2	0.8
a^I_3	0.5
:	:

$$priority(a^I_i) > priority(a^I_j) \Leftrightarrow i < j$$

6.3.4 Combining Agent-Dependent Feasibilities and Gain

While the above considerations concerning priorities of actions are based on the assumption that each robot can perform all actions we will consider the feasibility of actions *and* their gain (or priority) in the following. A number of examples show that we cannot ignore the feasibility of an action: For instance, a soccer robot without ball cannot perform the action shoot2goal, a robot that is stuck cannot move to a distant target position, and a robot short of energy resources has to move back to its home location. Recapitulatory this means that for certain applications the use of a priority list $\mathcal{A}^I_{pr}$ and a feasibility function $Q^{I\!I}$ is necessary.

In the following algorithm $\mathcal{A}^I_{pr}$ $(\hat{\mathcal{R}})$ denotes the priority list of actions (the set of idle robots). This algorithm takes into account that the actions are not equally important and not necessarily feasible for all robots.

$\hat{\mathcal{R}} = \mathcal{R}$
for $i = 1$ *to* $|\hat{\mathcal{R}}|$
    $\pi(r_i) =$ *no_operation*
for $i = 1$ *to* $|\mathcal{A}^I_{pr}|$
    if $\hat{\mathcal{R}} \neq \emptyset \wedge \max_{r \in \hat{\mathcal{R}}}(Q^{I\!I}(r, a_i)) > 0$
        $r_j = \arg\max_{r \in \hat{\mathcal{R}}}(Q^{I\!I}(r, a_i))$
        $\pi(r_j) = a_i$
        $\hat{\mathcal{R}} = \hat{\mathcal{R}} \setminus r_j$

The algorithm initializes all robots with the action *no_operation* and then loops over all actions in the order of their priority. If there is an idle robot left that can perform the current action the robot with the maximal feasibility to perform this action is chosen to do so. The feasibility $Q^{I\!I}(r, a)$ is set to zero if a robot r cannot perform the action a and to a value dependent on the predicted cost $\mathcal{C}(r, a)$ otherwise:

$$Q^{\Pi}(r,a) = \begin{cases} 0 & \textit{if} \quad \text{robot } r \text{ cannot perform action } a \\ \frac{1}{1+\mathcal{C}(r,a)} & \textit{otherwise} \end{cases} \tag{6.13}$$

Herein the predicted cost $\mathcal{C}$ depends on the application and its specific definition of cost. In section 6.4.1 we will see an example for the representation of $\mathcal{C}$.

6.4 Prediction-Based Coordination of Multiple Soccer Robots

Multiple collaborating robots with a common goal have to coordinate their actions in order to avoid physical interferences and to achieve a maximum of speed-up. Reasonably, in cooperative multi robot systems the common goal is decomposed into several tasks related to the individual robots of the system. The decomposition in terms of tasks is unique and changes depending on the current situation. Assuming such tasks can be performed by a single action each the question is how to assign tasks (=actions) to the robots. Actually a sequence of actions is required for each robot to achieve the common goal. Typical multi robot applications dealing with cooperative action selection include robot soccer, exploration, mine sweeping, messenger systems, and more. For an appropriate action assignment the cost $\mathcal{C}(r,a)$ of a robot r executing an action a must be well known. This requires an accurate prediction of the cost.

Assuming there is a common basis for action selection (see section 6.3.1) that characterizes the current state of the whole multi robot system the question remains how to best coordinate the robots' actions. Robots must reason about expected team utilities of future team states [Tambe and Zhang 1998]. In general, forestalling interferences and estimating the team utility requires the prediction of the consequences of a particular action assignment for the robot team.

To coordinate a team of multiple robots with a common goal in a shared environment we use a layered hybrid system containing a state estimation module, an action selection unit, and a multi robot navigation system. The hybrid architecture works as follows (see fig. 6.3). The robot's *state estimation* is based on *local sensor data* and *key features* from the other robots of the system (if available via communication). Local key features are sent to the other robots. The robot estimates the environment's state from its local view and from the views of the other robots. Cooperative *action selection* is done by the *utility function* that relies on all of the robots' *local views*. A neural *projector* supports the utility function by predicting the time need for desired changes in state. Thereafter the target states of the actions are given to the *multi robot navigation system* that considers path planning and computes an appropriate *low level control command* for the robot.

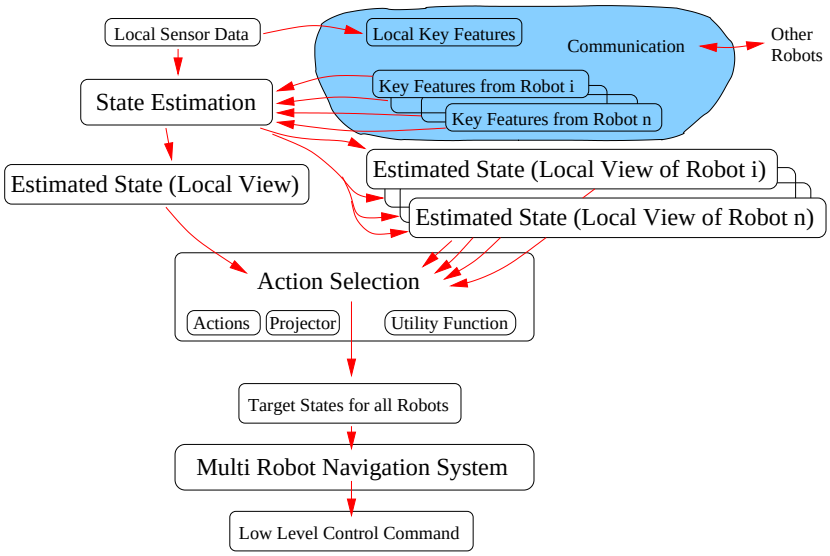


Figure 6.3: A robot architecture in a multi robot system to facilitate cooperative behavior.

This technique enables the action selection unit of a robot to put the robot in another robot's situation, and thereby, to consider the choice of the respective robot. This consistency allows the robots to behave cooperatively and avoids robots interfering one another. Action selection is done by the utility function (described by the algorithm in section 6.3.4) which is based on a priority list of actions. It is supported by a sophisticated neural prediction system (described in section 6.4.1) that estimates the time need for a requested change in state. Actions are assigned at a frequency of 10 Hz and may change depending on the current situation even if the respective task was not completed. There is *no explicit synchronization* between the robots but each robot deciding every $100ms$ allows only for very short times of double assignments.

Using the algorithm described in section 6.3.4 each robot employs the function $\mathcal{C}$ to predict its own cost and to compare it with the values computed for its team mates. Relying on identical software and the same local environment data the action selection of this algorithm is unique and consistent. $\mathcal{C}$ can already rely on the knowledge which robots are to perform the more important actions by predicting the cost for robot r_i performing a less important action a_j. In case there is any interference $\mathcal{C}(r_i, a_j)$ will increase. This mechanism forces cooperative behavior and informs all robots on the action selection of their team mates. The major advantages of this approach are

- A robot can put itself into the situation of a team mate. Therefore it will behave cooperatively by considering the team mates' decisions.

- All sequences of chosen actions can be stored locally and toggling situations in action selection can easily be detected and avoided.

- The algorithm is resistant against a crash of a single robot. Such an incident will be detected (for instance if the robot is not moving *and* not communicating for a certain time). Likewise a new robot can easily be integrated in the system. Systems using a high-accuracy state estimation will even work if there is no communication.

- The concept works fine with homogeneous and heterogenous robots. In case of heterogenous robots $\mathcal{C}$ must be implemented for each different robot.

- The laborious computation of a global map of the environment is not necessary. There is no loss of time for computing and broadcasting a global map.

6.4.1 Prediction of Time Needs

To improve the competence of our robot soccer team we have learned a model in order to predict the time need for performing a given single robot navigation task. To learn such a model with an expected inaccuracy of less than three percent we had to collect data from more than 10000 navigation episodes. Assuming that setting up and executing a navigation task takes only two minutes we would have had to spend more than 330 hours of experimentation with the real robots (see the example in chapter 3 too). Obviously, this is not feasible.

As stated in section 6.3.4 the cost $\mathcal{C}(r_i, a_j)$ (in equation (6.13)) for robot r_i performing action a_j depends on the specific application. In traveling it may be the way, in mine sweeping it may be the time need per mine, and in robot soccer it can be the time to score or prevent a goal. In our case the cost $\mathcal{C}(r_i, a_j)$ can be described as the time robot r_i needs to complete action a_j. Assuming that to complete an action means to reach a certain target state we try to predict the time a robot needs to complete a single robot navigation task. This will give us an information how suitable it is that this robot performs the task proposed. This means we have to estimate the time need for a robot completing an action considering the following:

(1) Knowledge about the dynamical behavior of the robot must be acquired.

(2) Actions of team mates must be taken into account.

(3) Other dynamic objects must be regarded as moving obstacles.

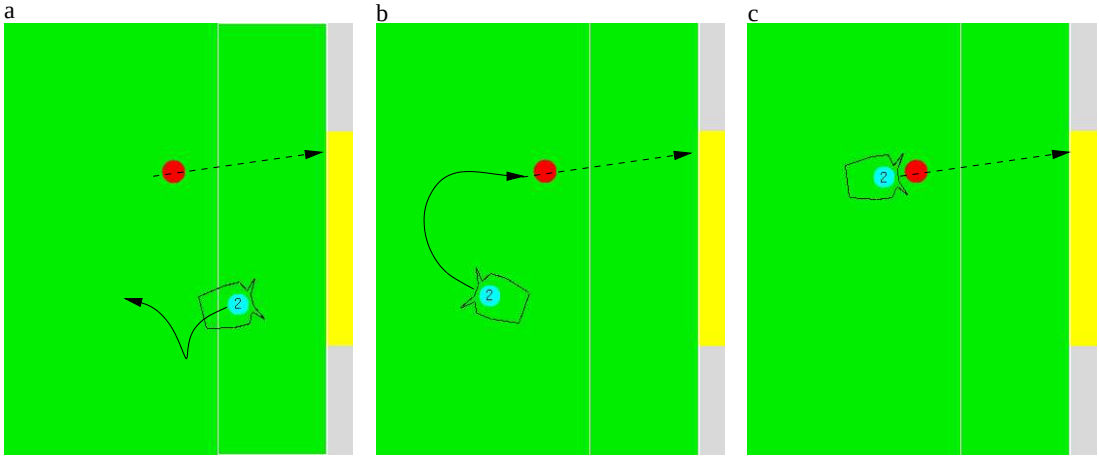


Figure 6.4: A training scenario in the multi robot simulation environment.

Due to the physical complexity of this problem it seems impossible to estimate the amount of time without learning algorithms. But, once again, learning the prediction

$$\mathcal{C} : \mathcal{R} \times \mathcal{A} \rightarrow time \tag{6.14}$$

with data from real robot runs would require an impractical huge amount of time for data acquisition. It is less expensive to use the robot simulator described in section 3.5.1 and in [Buck *et al.* 2002c]. This simulator mimics the physical behavior of the robots.

Learning a Neural Prediction Neural Networks have been shown to be an accurate means for the prediction of run-times (see [Smith *et al.* 1999] for example). Hence we choose neural learning to obtain $\mathcal{C}$. We apply multi layer neural networks (see appendix A.2) and the backpropagation derivative RPROP [Riedmiller and Braun 1993] because of its adaptive step-size computation in gradient descent. The data we use to train the neural prediction $\mathcal{C}$ is completely obtained from the learned simulator in minimal time. The training patterns have the form

$$\langle \zeta_s, \zeta_t \rangle, time \tag{6.15}$$

where ζ_s is the randomly chosen start state of the robot and ζ_t its randomly chosen target state in the simulation. We get the necessary value for *time* by simulating a robot driving from ζ_s to ζ_t (see fig. 6.4): The simulated robot is set to ζ_s (position of the robot in subfigure (a)) and has to drive to the target state ζ_t indicated by the dashed arrow. The direction and length of this arrow indicate

the target state's orientation and velocity. The time the robot needs to reach its target state (subfigures (b) and (c)) is taken to complete the training pattern (equation (6.15)). $\mathcal{C}$ was trained with around 300.000 patterns using a network with input layer, output layer, and two hidden layers. At learning time there are no other objects or team mates taken into consideration. Using validation patterns for early stopping [Sarle 1995] the trained network achieved an average error of 0.13 seconds per prediction on a test set not used for learning. Due to the complexity of this learning problem this is an appropriate result.

Taking the Actions of Team Mates into Account If we use the algorithm proposed in section 6.3.4 in order to select a robot to execute a particular action a_j we can consider the behavior of all robots executing actions a_i under the condition that $i < j$ (a_i prior to a_j). This is possible because we compute $\mathcal{C}(r, a_j)$ for any robot $r \in \hat{\mathcal{R}}$ dependent on the intentions of all robots performing actions a_k ($k \leq j$). For these robots, a set of start states and target states as well as the priority of each target state (action) is given. A multi robot navigation system can receive this data and computes the paths for all concerned robots including r. This multi robot navigation system consists of three components:

(1) The neural network controller described in section 5.5.

(2) A library of software tools for planning and plan merging including the methods presented in sections 5.6.2 and 5.6.3.

(3) The learned decision tree described in section 5.6.5 that selects the appropriate planning methods in a situation-specific way.

If the multi robot navigation system proposes to apply a path planning method for r to get from its start state ζ_s to its target state ζ_t and the first intermediate state on the computed path is ζ_i the time need is set to

$$\mathcal{C}(\zeta_s, \zeta_t) = \mathcal{C}(\zeta_s, \zeta_i) + \mathcal{C}(\zeta_i, \zeta_t) \tag{6.16}$$

This equation may be used recursively for the computation of $\mathcal{C}(\zeta_i, \zeta_t)$ if ζ_i is not the last intermediate state of the computed path.

6.4.2 Cooperative Action Selection

The algorithm described in section 6.3.4 has been implemented and tested in simulation as well as in a real robot environment. Being highly reliant on cooperation, soccer robots are a suitable appliance for testing this algorithm. We observed (1) the behavior of a team of 11 autonomous soccer robots in the RoboCup simulation league environment and (2) the behavior of four real robots belonging to the RoboCup mid-size league.

Communication	Match. rate pass player/receiver	Average goals per game
full	82.1%	8.1
breakdown each 120 s	76.2%	7.4
breakdown each 60 s	70.1%	6.5
breakdown each 30 s	66.1%	6.1
no communication	63.9%	5.7

Table 6.1: Average goals per game and matching rates for pass play depending on different communication modes.

Simulation Experiments

In the RoboCup simulation environment a soccer team consists of 11 autonomous software agents communicating with a server program [Kitano *et al.* 1997, Noda *et al.* 1998]. The agents receive sensory data and send low level control commands. We use the *Karlsruhe Brainstormers* agent of RoboCup 1999 [Riedmiller *et al.* 1999] as a basis for our experiments. Each agent can choose from a set of actions

$$\mathcal{A}^I = \{shoot2goal, pass2player, dribble, receive\_pass, go2ball, offer4pass, gohome\} \tag{6.17}$$

$Q^I(r,a)$ is represented by a priority list (see section 6.3.4). The feasibility $Q^{I\!I}(r,a)$ is instantiated by hand-coded functions based on situation dependent features. For action selection thresholds are used (see section 6.2.2). Actions are chosen at a frequency of 10 Hz. There is no direct communication between the agents but a limited communication via the soccer server simulation program. Exploiting this communication, the agents share their current perceptions of the system's state.

Pass Play We played several games and recorded the locally selected actions of all agents. We measured how many times the agent chosen to receive a pass by the agent playing the pass was identic with the agent planning to receive a pass in relation to all passes played. Herein, we rated a match within a temporal difference of 0.1 seconds (1 simulation cycle) as positive. This quotient gives us a measure of cooperation since pass play is a paradigm of cooperation. The experiments were performed with working communication and temporary disordered communication. In table 6.1 one can see the above explained quotient with working communication and with a communication breakdown every 30, 60, and 120 seconds which lasts t_{bd} seconds with t_{bd} uniformly distributed over $[0s, 30s]$. Furthermore we played games without any communication. Each result is based

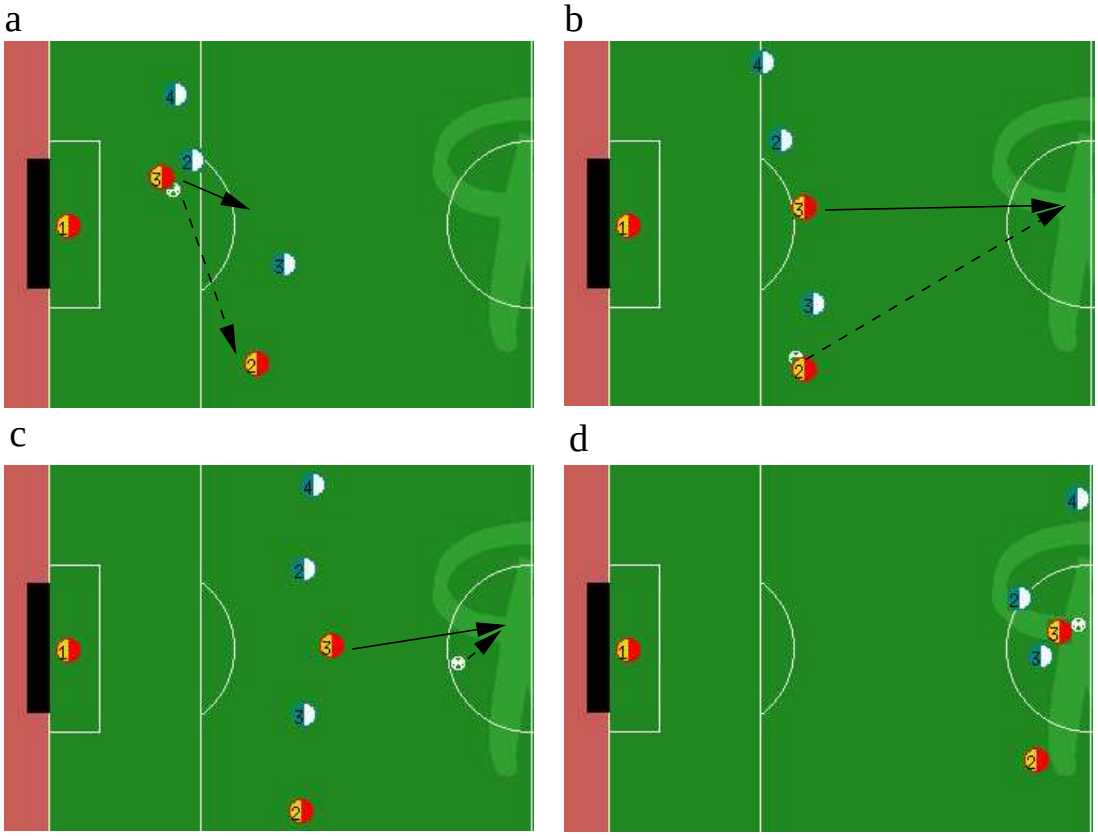


Figure 6.5: A double pass scenario in the RoboCup soccer server environment.

on 10 games respectively. One can see that an increasing frequency of communication breakdown accompanies with a decreasing number of scored goals per game. Moreover less communication means less successful planning of pass play and with it less coordination. Nevertheless a team using no communication is able to score around 70% of the goals a team with full communication scores. Additionally 77.8% of the matching quotient of a team using full communication is reached by a team without any communication. These results document that our algorithm for action selection and coordination is robust and can deal very well with temporary failures.

Double Pass Play Without ever explicitly learning to play a double pass and no special plan for a double pass specified not seldom double pass play was observed. Analyzing some cases of double pass play we found a reasonable pattern to explain this effect (see fig. 6.5). Robot number 3 holds the ball and decides to pass to player number 2. Meanwhile player number 2 puts itself in the situation of player number 3 and recognizes that it has to receive a pass (subfigure (a)):

$$\pi^I(r_3) = pass2player_2 \qquad \pi^I(r_2) = receive\_pass_3$$

Immediately after having played the ball player 3 performs $\pi^I(r_3) = \textit{offer4pass}$ to offer itself for a receipt of a pass. As player 2 receives the ball it chooses to play a pass to player 3. Meanwhile player 3 puts itself in the situation of player number 2 and recognizes that it has to receive a pass (subfigure (b)):

$$\pi^I(r_2) = \textit{pass2player}_3 \qquad\qquad \pi^I(r_3) = \textit{receive\_pass}_2$$

Further player 3 performs $\pi^I(r_3) = \textit{receive\_pass}$ while player 2 performs $\pi^I(r_2) = \textit{offer4pass}$ after playing the pass back to player 3 (subfigure (c)). Receiving the ball again player 3 is to choose from $\{\textit{shoot2goal,pass2player,dribble}\}$ again (subfigure (d)).

Real Robot Experiments

To evaluate our approach in a real robot environment we choose the RoboCup mid-size league (see appendix B about RoboCup). Two teams of 4 players compete on a field of about 9 meters in length and 5 meters in width. Compared with the RoboCup simulation league there are a number of new challenging problems: The sensory data is not provided by a server program but must be acquired by the robot itself. Furthermore most robots are not able to receive a pass from any direction unlike in the simulation league. Further, path planning becomes more important on a comparably small field (the field in the simulation league is 105 meters long). For our experiments we use the Agilo RoboCuppers team (appendix B.2,[Beetz *et al.* 2002]) as a basis. As stated above these robots have to acquire information about their environment autonomously. In the following we will shortly describe the methods employed for state estimation before we regard the experiment itself.

Probabilistic State Estimation We employ the state estimation module for individual autonomous robots described in [Schmitt *et al.* 2001a] that enables a team of robots to estimate their joint positions in a known environment (such as a soccer field or an office building) and track the positions of autonomously moving objects. The state estimation modules of different robots cooperate to increase the accuracy and reliability of the estimation process. In particular, the cooperation between the robots enables them to track temporarily occluded objects and to faster recover their position after they have lost track of it.

The state estimation module of a single robot is decomposed into subcomponents for self-localization [Hanek and Schmitt 2000] and for tracking different kinds of objects. This decomposition reduces the overall complexity of the state estimation process and enables the robots to exploit the structures and assumptions underlying the different subtasks of the complete estimation task. Accuracy and

#robots performing *go2ball* at the same time	quota in relation to the whole time played
0	00.34%
1	98.64%
> 1	01.02%

Table 6.2: The number of robots performing *go2ball* at the same time.

reliability is further increased through the cooperation of these subcomponents. In this cooperation the estimated state of one subcomponent is used as evidence by the other subcomponents.

Considering further Physical Properties for State Estimation So far physical properties like dead time and acceleration of robots are not considered for the state estimation. All robots are dealing more or less with a dead time which means that at the moment a robot is performing visual localization it has already sent control commands (e.g. an action a to control translational and rotational velocity) for further movements. These sent commands can be used to predict the robot's state a little time ahead. This requires a mapping

$$\mathcal{P} : \zeta_c \times a \mapsto \zeta_{succ} \tag{6.18}$$

from a current state ζ_c and a sent command a to the successor state ζ_{succ}. This mapping is a special case of the mapping in equation (3.7), chapter 3. $\mathcal{P}$ is learned from experience, as described in section 3.5.1. Depending on the number of command cycles equating with the dead time, $\mathcal{P}$ has to be applied repeatedly to the current state. This enables a robot to perform action selection and path planning considering its future state which is predetermined anyway.

In many mobile robot applications the machines can get stuck or blocked by other objects. It is very useful to detect such a situation because it might strongly change the process of action selection, for the robot affected and for its team mates. To detect such a situation we employ $\mathcal{P}$ once again. Recording the low level commands sent to the robot we can predict the robot's changes in state. If they differ significantly from the measured changes in state we assume the robot is not able to move correctly. This information is posted among the robots of the team and will be considered by the action selection unit.

Approaching the Ball In the following the list of actions theoretically available for each robot of the team is defined by

$$\mathcal{A}^I = \{shoot2goal, dribble, clear\_ball, go2ball, gohome, get\_unstuck\} \tag{6.19}$$

Action selection is supported by thresholds (see section 6.2.2 for details). $Q^I(r,a)$ is represented by a priority list (see section 6.3.4). The feasibility $Q^{II}(r,a)$ is computed dependent on a learned cost function (equation (6.13), section 6.4.1). To demonstrate the coordination of the team we measure the number of robots performing *go2ball* at the same time. Further we observe how long the same robot performs *go2ball* without being interrupted by another robot. Here we rate only actions that last for more than one cycle ($> 200ms$) as an interruption. The data was acquired from five real robot games against different opponent teams at the international robot soccer world cup 2001. Table 6.2 depicts in how many percent of the whole time played no robot, one robot, or more than one robot performed *go2ball*. The frequency of action selection is around 10 Hz. The average time one robot performs *go2ball* or handles the ball without being interrupted by a decision of a team mate is 3.35 seconds. In only 0.25% of the time a robot that is stuck is determined to go for the ball by the other robots. However, the positioning of the robots supports the surprisingly good results of this experiment: (1) Normally, the robots' positions are far from each other. This leads to less conflicts in decision making. (2) If a robot has the ball in its guiding device no other robot will decide to go for the ball.

Nevertheless, these results show that, in the context of robot soccer, coordination can be achieved to a great extent. There are hardly ever situations where no robot or more than one robot approaches the ball at a time and the situations in which it is not clear which robot is to go for the ball are just a few.

Solving Situations with Stuck Robots Not seldom, in a highly dynamic environment like robot soccer a robot gets stuck due to another robot blocking it. The following example shows how such incidents were handled by the robots in our experiments (see fig. 6.6). Robot number 2 is supposed to be the fastest to get the ball and therefore approaches the ball (subfigure (a)):

$$\pi^I(r_2) = go2ball$$

Near the ball robot 2 collides with an opponent robot. Robot 2 is in a deadlock situation and cannot move forward anymore. The only action feasible to execute remains *get_unstuck*. Thus robot 3 approaches the ball now (subfigure (b)):

$$Q^{II}(r_2,a) = 0 \; \forall a \in \mathcal{A} \setminus get\_unstuck \quad \pi^I(r_2) = get\_unstuck \quad \pi^I(r_3) = go2ball$$

Having reached the ball robot 3 dribbles towards the opponent goal while robot 2 is moving backwards (subfigure (c)):

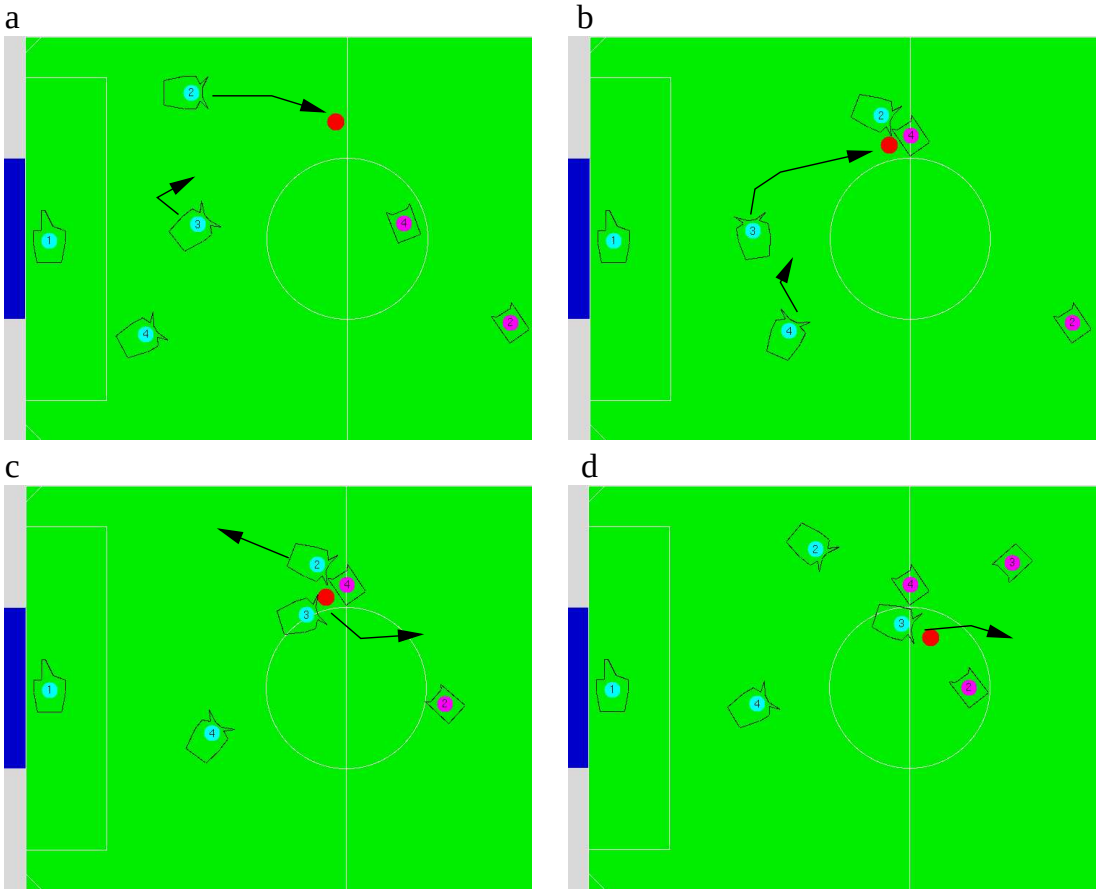


Figure 6.6: An example for intelligent cooperation in a real robot soccer environment.

$$Q^{II}(r_2, a) = 0 \; \forall a \in \mathcal{A} \setminus \textit{get\_unstuck} \quad \pi^I(r_2) = \textit{get\_unstuck} \quad \pi^I(r_3) = \textit{dribble}$$

Further on robot 2 is no more stuck and robot 3 is still dribbling (subfigure (d)):

$$Q^{II}(r_2, \textit{go2ball}) > 0 \qquad \pi^I(r_2) = \textit{gohome} \qquad \pi^I(r_3) = \textit{dribble}$$

Finally the feasibility for robot 2 performing *go2ball* is not zero any more but since robot 3 has the ball robot 2 chooses another action.

The above experiments show that the ability of a robot to put itself in its team mate's situation supports cooperative behavior among the robots. It is shown that the proposed approach is reliable and, to a high extent, fault tolerant even if there is no communication between the robots but a reliable localization. The neural prediction (section 6.4.1) to estimate the time need for a robot itself or for other robots to reach a certain state is extremely helpful to determine the feasibility of an action. Compared to most previous methods our concept described

in section 6.3.4 is neither leader-following nor dependent on a shared global map of the environment. Each robot decides completely autonomous supported by a neural prediction. In action selection we consider the feasibility of an action as well as its priority.

Chapter 7

Related Work

In this chapter, we consider previous research that aims at goals similar to ours and put it in relation with our work.

7.1 Reinforcement Learning

The idea of automatic learning based on experience has been surveyed by many researchers. In relation to the algorithms described in this work, the techniques of reinforcement learning are of special interest. Reinforcement learning[1] can be characterized by *learning from interaction with the environment using a trial and error search with feedback through a delayed reward*. The goal of reinforcement learning is to obtain a policy-function π that maps from a situation to an appropriate action. Further, all common reinforcement learning algorithms rely on finite Markov decision processes.

Markov Decision Processes

Markov decision processes (MDPs) [Ross 1983, Bertsekas 1987] are a fundamental basis for reinforcement learning. They provide a model for actions, states, and transitions between them. Finite MDPs consist of the following components:

$$\text{MDP} = \langle \mathcal{S}, \mathcal{A}, P, R \rangle \tag{7.1}$$

$\mathcal{S}$ is a finite set containing all states a machine can be in. A state $\zeta \in \mathcal{S}$ must provide all information that is necessary to choose the optimal action in order to have the Markov property. $\mathcal{A}$ is a finite set of actions from which actions can

[1] [Bertsekas and Tsitsiklis 1996, Sutton and Barto 1998] give a good introduction to reinforcement learning

be chosen to be executed depending on the current state. P denotes probability distributions for changes in state. For instance, $P(\zeta(t), a, \zeta')$ gives the probability for a transition from state $\zeta(t)$ to state ζ' if choosing action $a \in \mathcal{A}$. $R(t+1)$ represents the immediate reward the learning algorithm obtains for executing action a in state $\zeta(t)$.

7.1.1 Temporal Difference Learning Methods

As a central method in reinforcement learning, temporal difference learning [Sutton 1988] incrementally updates the estimates of a value-function or a Q-function partly based on existing estimates and partly based on immediate rewards. The success of this strategy is documented by a number of applications such as Tesauro's learned backgammon player [Tesauro 1994] that was among the best players of the world. The basic TD(0) algorithm incrementally updates a value-function over the state space using the following equation:

$$\mathcal{V}(\zeta(t)) = \mathcal{V}(\zeta(t)) + \alpha[R(t+1) + \gamma\mathcal{V}(\zeta(t+1)) - \mathcal{V}(\zeta(t))] \tag{7.2}$$

Herein α is a learning rate tuning the impact of the immediate reward R and the existing estimate of the successor state $\zeta(t+1)$. γ is a discount factor tuning the influence of the existing estimate of the successor state only. If we want to exploit a learned value-function actions are chosen by the policy:

$$\pi(\zeta(t)) = \arg\max_{a \in \mathcal{A}} R(t+1) + \gamma \cdot \sum_{\zeta'} P(\zeta(t), a, \zeta') \cdot \mathcal{V}(\zeta') \tag{7.3}$$

SARSA Instead of a value-function, SARSA [Rummery and Niranjan 1994] learns a Q-function depending on state *and* action. The resulting equation (of the basic SARSA) for updating the Q-function is given by

$$Q(\zeta(t), a(t)) = Q(\zeta(t), a(t)) + \alpha[R(t+1) + \gamma Q(\zeta(t+1), a(t+1)) - Q(\zeta(t), a(t))] \tag{7.4}$$

Unlike in the following Q-learning, this update is dependent on the Q-value for $\zeta(t+1)$ and $a(t+1)$ that must be defined in advance. [Stone and Sutton 2001] have applied the SARSA(λ) algorithm to multiple agents in the RoboCup soccerserver simulation environment [Kitano *et al.* 1997, Noda *et al.* 1998] and learned playing behaviors for a 2 vs. 3 scenario.

Q-Learning The Q-learning algorithm [Watkins and Dayan 1992] directly approximates the optimal Q-function. It is by far the most applied technique in reinforcement learning. Like SARSA, Q-learning incrementally learns a state-action Q-function from experience. $Q(\zeta, a)$ computes how good it is to perform action a in state ζ. According to

$$\begin{aligned} &Q(\zeta(t), a(t)) = \\ &Q(\zeta(t), a(t)) + \alpha(R(t+1) + \gamma \max_a Q(\zeta(t+1), a) - Q(\zeta(t), a(t))) \end{aligned} \tag{7.5}$$

Q is updated incrementally where γ is a discount factor tuning the impact of the maximized Q-value of the successor state. $R(t+1)$ is the immediate reward received for performing action $a(t)$ in state $\zeta(t)$. In order to exploit a given Q-function we use the following policy

$$\pi(\zeta(t)) = \arg\max_{a \in \mathcal{A}} Q(\zeta(t), a) \tag{7.6}$$

for choosing an appropriate action. A lot of successful work has been done with Q-learning: [Asada *et al.* 1996] has transformed camera images into discrete states and applied Q-learning in that space. In further work, [Takahashi and Asada 2000] applied vision-guided multi-layer Q-learning where the robot autonomously determines subtasks without the intervention of a human developer. Low-level skills and 2 vs. 1 behavior have been learned in the RoboCup simulation environment [Riedmiller and Merke 2001]. Navigation tasks [Lin 1992] and multi robot cooperation [Mataric 1997] have been learned with Q-learning. [McCallum 1995] has learned a highway driving task including other cars and overtaking scenarios. [Forbes 2000] has used Q-learning for autonomous vehicle control. [Martinson *et al.* 2002] have learned a high-level behavior selection using Q-learning, and composition policies for basic controllers have been learned too [Huber and Grupen 1997].

Eligibility Traces As an extension for all temporal difference learning methods, [Watkins 1989] introduced the use of eligibility traces. In this mechanism, rewards are not assigned immediately after each action (1-step algorithms) but backups after a certain number of steps are used for learning.

Dyna-Q and Prioritized Sweeping In addition to Q-learning, in the Dyna-Q architecture [Sutton 1990] a model of the environment is acquired during 1-step Q-learning and exploited for simulated experiences made in parallel to observed real experiences. The environment is assumed to be deterministic, and 1-step simulations are made with state-action-pairs visited previously only. Prioritized sweeping [Peng and Williams 1993, Moore and Atkeson 1993] is based

on the idea that the values of predecessor states of those states whose value has changed a lot are more likely to change also a lot. Thus the simulated experiences of Dyna-Q are directed to those states whose values have changed most.

7.1.2 Recent Improvements

Grid Size of Discretization In order to both, limit the number of discrete states and smooth the control output, [Moore and Atkeson 1995] propose to increase the resolution of the state space in interesting regions while decreasing it in less interesting regions. [Kondo and Ito 2002] propose to divide the state space gradually according to the progress of learning.

Continuous Approximations of the Value-Function One standard approach for reinforcement learning in continuous state spaces is to use the gradient of the value-function to choose an action [Werbos 1990]. But only replacing the discrete value-function by a continuous approximation has been shown to fail [Boyan and Moore 1995]. [Thrun and Schwartz 1993] find the *overestimation phenomenon* to be the main reason for that. Overestimation results from noise which is likely in real world applications. Since temporal difference methods are incremental algorithms there is the danger of an accumulation of such errors. Recent work on approximating a continuous value-function in reinforcement learning include the HEDGER algorithm [Smart and Kaelbling 2000], barycentric interpolators [Munos and Moore 1999], and instance-based learning [Forbes and Andre 2000]. [Takahashi *et al.* 1999] introduced continuous valued Q-learning where some kind of interpolation between discrete states is performed based on contribution vectors. Another problem reported in the literature is the discontinuity problem [Takeda *et al.* 2001]. It addresses the question what happens if the optimal value-function (or Q-function) is discontinuous. Nearly all common function approximators will fail here. Happily, in practice this case is not of major interest because most value-functions are continuous.

Hidden State Traditional reinforcement learning assumes that all factors relevant for the computation of the value-function (or Q-function) are observable. But in practice this constraint might fail. Due to unobservable parts of the state space this problem is called *hidden state*. For example, in the dribble-task in section 5.4 there is hidden state, namely the perception of the defending robot. In this application we overcome the hidden state by a specific model for the hidden part. In general, such a model is not provided. [McCallum 1995] has developed successful solutions to deal with hidden state.

7.1.3 Comparing Reinforcement Learning and EBC

The methods of *experience-based control* (EBC) introduced in this dissertation, on one hand, share a lot with reinforcement learning while, on the other hand, we can delineate quite a few differences between the two approaches.

Common Background of Reinforcement Learning and EBC Obviously, reinforcement learning and EBC both exploit experiences from interaction with the environment. The key idea for both approaches is to use feedback gained from exploration to improve the controller's performance. Exploration is implemented as a trial and error search in the action space.

Differences between Reinforcement Learning and EBC In EBC learning is performed *directly in a continuous state space*. To apply temporal difference learning methods for tasks in continuous domains like machine control the state space is usually discretized [Kalmar *et al.* 1998]. But in general, operating in a discrete state space brings some well known problems: Using a coarse discretization the control output is not smooth and using a fine discretization the number of states becomes huge, especially in high-dimensional spaces [Doya 2000].
The *non-incremental learning* in EBC avoids the summation of errors which is a frequent problem in temporal difference learning. Moreover old information is not forgotten. On the other hand, EBC is not able to keep adapting to changes in the environment lifelong.
So far, in reinforcement learning the danger of unsuccessful exploration[2] has been widely neglected. In EBC, we overcome this negligence by introducing the *backward exploration* that guarantees to find trajectories that lead to the target state. One might regard robot shaping [Dorigo and Colombetti 1994, Perkins and Hayes 1996] as a related approach for reinforcement learning. In robot shaping the machine starts to explore close to its target state first. In the following, the start state is incrementally moved further away from the target state.
In reinforcement learning three parameters, namely the reward, the discount factor, and the learning rate have an impact on the approximation of the value-function (or the Q-function). In EBC we employ a *discount factor γ only*. There is no reward at any time. The discount factor tunes the relation between consecutive states (or state-action pairs).
Reinforcement learning relies on MDPs that work with probability distributions. State spaces in machine control are likely to include some nonlinearities resulting from angles, acceleration, etc. These nonlinearities are hard to discretize

[2] In chapter 4 unsuccessful exploration is defined as an exploration where the target state has not been reached.

reasonably. It is extremely difficult to determine the neighborhood of a state accessible by any action in a high dimensional state space including nonlinearities. Therefore it may be difficult to apply MDPs for certain problems. In EBC, the projection-function $\mathcal{P}$ is learned from experience (see section 4.2) and replaces the probability distributions P of the MDP. *No* MDP *model is needed at all.* This is possible because in EBC we make the assumption of a deterministic environment or at least of an unimodal distribution with a small variance for the impact of actions. In fact, most real world machine control tasks suffice these requirements[3]. However, in highly stochastic applications like games (throwing dice etc.) the use of a deterministic projection-function will not work.
The consideration of *undesirable states*, and especially negative terminal states, for reinforcement learning has received surprisingly little attention in the past. In EBC, undesirable states can easily be integrated in the value-function approximation. [Geibel 2001] has worked on the integration of fatal states in reinforcement learning.
While reinforcement learning methods like Q-learning, Dyna-Q, and others are theoretically designed under the assumption that there is unlimited time for exploration, in EBC we perform an inductive approximation of a distance metric that can *generalize even from very few patterns.*
Furthermore, EBC provides methods to *directly learn the policy-function.* EBC reliably solves complex machine control problems in highly dynamic, high-dimensional, and continuous state spaces based on a very small number of explorations even if exploration remains unsuccessful using conventional techniques.

7.2 Conventional Control Methods

In this section, we briefly consider PID controllers and fuzzy control and put it in relation with the control methods of EBC. For more details on conventional controllers see [Jacobs 1993] for example.

7.2.1 PID Control

For many practical applications, the well surveyed PID controller is used. Here, we go briefly over its main characteristics to compare it with the methods of EBC. PID stands for *proportional*, *integral*, and *differential.* The controller consists of three components that have different objectives:

$$a = K_R(E + \frac{1}{T_N}\int_0^t E d\tau + T_V \cdot E') \tag{7.7}$$

[3]Our preceding experiments in chapter 5 underline this.

The first component, the proportional part of the controller, adjusts the action a proportional to the error E weighted by the factor K_R. The integral component increases the accuracy of the controller. It is weighted by $\frac{K_R}{T_N}$. Finally, the third component, the differential part, aims at a fast control of the system. $K_R \cdot T_V$ weights the differential part. Using only three parameters, this controller can implement a wide variety of control behavior. This makes it one of the most used controllers in practice. However, due to its structure, the PID controller assumes the state-value to be monotone towards the target state in all dimensions. Inherently, control problems where the state-value of a particular dimension behaves not monotonic cannot be solved. In EBC, we learn a problem-specific distance metric to overcome this problem.

7.2.2 Fuzzy Control

In contrast to a PID controller, the transfer behavior of a fuzzy controller is not implemented explicitly but implicitly by means of characteristic diagrams. The action depends on the errors in the single dimensions of the state space:

$$a = f(E_1, E_2, \ldots) \tag{7.8}$$

Therefore cases where the state-value is not monotone in all dimensions can be solved. The characteristic diagrams are represented by fuzzy-sets and rules. However, the rules are usually designed based on a priori knowledge. Therefore, the complexity of the control problem should be known in advance. Fuzzy controllers have been used for technical applications for a long time now. Two of numerous examples are warm water plants [Kickert and van Nauta Lemke 1976] and train operation systems [Yasunobo and Mamdani 1985]. For a good overview on fuzzy control see [Cox 1992, Driankov *et al.* 1993, Graham and Ollero 1996]. Like fuzzy controllers, EBC relies on characteristic diagrams. In EBC, they are represented by a learned value-function for example. Therefore, without any prior knowledge, it is much more feasible to develop controllers in EBC than in fuzzy control.

7.3 Robot Control

Over the past ten years, a number of impressive demonstrations have shown that robots controlled by artificial intelligence are already able to solve tasks in real world environments. The vehicle ALVINN is a neural controlled small truck trained with data from human drivers [Pomerleau 1993, Sukthankar *et al.* 1993]. In extensive tests it proved a high reliability. The autonomous mobile robots Rhino [Thrun *et al.* 1998] and Minerva [Thrun *et al.* 2000] successfully guided

thousands of visitors through museums in the USA and in Germany. The RoboX system consisting of ten autonomous interactive robots worked as a tour guide at the swiss Expo.02 [Jensen *et al.* 2002]. Besides sensor integration and cognitive interaction, path planning in crowded and highly dynamic environments is one of the fundamental computational problems of such robot control systems.

7.3.1 Path Planning

Besides the methods for robot path planning introduced in sections 5.6.2 and 5.6.3 there exists a wide variety of algorithms. For a general introduction to path planning [Latombe 1991] is recommended. One of the keys for the successful application of path planning methods to real robots is the consideration of the robot's dynamical properties. While in section 5.6 we learn to choose the appropriate planning methods dependent on the current situation without using explicit knowledge about the robots' velocities, some algorithms explicitly include the robots' velocities in their planning process. Here, recent work includes the dynamic window approach (DWA) [Fox *et al.* 1997, Brock and Khatib 1999]. In this method, a trajectory in the velocity space of the robot is planned considering the robot's dynamic constraints. A linear evaluation function chooses the velocities to be set weighing clearance, speed, and the heading towards the target. [Belker and Schulz 2002] propose an extension of the DWA using optimal utility functions. [Minguez *et al.* 2002] extend standard path planning algorithms by integrating the robot's dynamics into the algorithms. [Bruce and Veloso 2002] propose the rapid exploring of random trees and apply it successfully to the RoboCup small-size league.

Multiple Robots In a nutshell one can classify multi robot navigation planning methods into ones that plan the joint navigation task in one shot such as [Leroy *et al.* 1999, Bennewitz and Burgard 2000] who operate in a combined configuration time space and ones that essentially decompose a multi robot navigation task into single navigation tasks and merge and repair the resulting plans in order to avoid conflicts with the navigation plans of other robots. For example, [Kant and Zucker 1986] propose an algorithm for navigation problems in dynamic environments that decomposes navigation planning into planning a collision free path with respect to the static obstacles and then determines the velocity along this path in order to avoid collisions with the moving obstacles. Unfortunately the application of such methods for planning in dynamic environments requires predictive models of the obstacles' movements, which in many applications cannot be provided for all obstacles because of likely frequent, abrupt, and unpredictable changes in the speed and in the orientation of the obstacles' movements. The situation dependent selection of planning methods using EBC described in section 5.6 was trained assuming random movement of the obstacles. This might

not lead to an optimal behavior but produces a defensive rather than a risky behavior. To obtain an optimal behavior, however, the laborious and sometimes even online computation of a model of the obstacles' movements is necessary.

7.3.2 Multi Robot Coordination

A number of tasks require multiple cooperating agents in order to be solved. The two different approaches of leader-following and behavioral (decentralized) schemes have already been introduced in section 6.3.1. In this work, we focus on the latter method because it is much more fault tolerant than a leader-following system. In the robot soccer environment, [Castelpietra *et al.* 2000] coordinate their robots by a greedy algorithm that assigns actions to all robots. While our soccer robots decide supported by neural predictions of the actions' cost, [Weigel *et al.* 2002] employ behavior networks for the action selection of their soccer robots. In this approach, actions are connected to, and triggered by, special features that are suitable to characterize the current situation. [Khoo and Horswill 2002] have developed the HIVEMind architecture for the coordination of robot groups and applied it to search tasks. All robots of the group share sensory data by treating other team mates' sensors and actuators as virtual parts of their own. [Jäger 2002] has successfully implemented a totally decentralized approach using communication only in a local neighborhood to coordinate multiple cleaning robots working in a supermarket. The M+ architecture [Alami and Botelho 2002] employs the CN-Protocol that puts up actions for auction among the robots with their respective utilities. M+ has been used for bed cleaning and object transfer in a hospital environment. A survey on recent work on multi-agent coordination is given in [Pynadath and Tambe 2002]. From a more theoretically point of view [Gerkey and Mataric 2002] state that multi robot task allocation can be reduced to an instance of the *Optimal Assignment Problem*. They compare different well known approaches considering their computational and communicational amount.

Chapter 8

Conclusions and Future Work

In this chapter, we summarize the scientific contributions of this dissertation and suggest promising directions for further research in this area.

8.1 Conclusions

In this dissertation, we have introduced algorithms for autonomous machine control that learn by induction from experience and interaction with their environment. The most important aspect was to develop algorithms that work reliably in real robot settings. Most of the proposed approaches have been developed in the context of robot soccer in order to work on real robots. In our work, learning algorithms are not only implemented for virtual environments. In fact, the physical behavior of real machines is considered as well. Thus our methods have been shown to run on real robots. The techniques have succeeded under competitive conditions, such as robot tournaments, without any intervention from a human user.

In chapter 1 we stated that we are interested in developing methods that are easy to implement and that work automatically to a large extent. The algorithms presented in this dissertation are indeed easy to implement as they require very little a priori knowledge of the problems to solve. In addition, the tuning of parameters is reduced to a minimum while remaining parameters such as the discount factor γ are not magic numbers but parameters whose impact on the learning process is well understood. The entire learning process can be automated by a program consisting of two subprograms, one for exploration and one for function approximation. In chapter 5 we have seen a number of different applications that have been solved successfully with the methods that are introduced in this dissertation. In extensive experiments we have applied different strategies for exploration in order to construct patterns for learning. These applications show

that our algorithms work reliably, accurately, and in real-time. We have studied tasks with high-dimensional state spaces where it is not sufficient to simply use the Euclidean or Minkowski distance metric for global optimization. We have seen that it is practicable to learn a nontrivial distance metric and to represent it by a value-function. The success of the proposed methods is founded in their technical contributions that we will refer to in the following.

The *curse of dimensionality* makes learning by induction in high-dimensional state spaces difficult: Unexplored regions of the state space must be represented in an appropriate way. In our approach, we can learn a value-function even from very few example patterns. If, for instance, there is only one successful exploration we are still able to learn a value-function from that data. However, in nearly all cases the more learning data is available, the better the performance will be. But in practice, the amount of learning data is limited by the time available for data acquisition. For complex learning tasks, it may take several minutes to perform just one experiment with a real machine. In this case it becomes impractical to perform thousands of experiments. To perform a big number of experiments it is inevitable to use simulation techniques. However, the simulation of a complex task is likely to fail. Therefore in chapter 3, we propose to decompose such simulations into sets of less complex subsimulations. For example, the simulation of a multi robot system is decomposed into a number of single robot simulations. The neural simulation of the dynamics of robots in section 3.5.1 has been shown to be accurate and has been used as a basis for more complex learning problems.

Since we rely on a limited amount of training data, we cannot guarantee that our algorithms will find the optimal solution to a problem. But the approximated value-function is guaranteed to be a lower bound of the optimal value-function if the employed function approximator works reliably. Furthermore, with an increasing number of successful explorations it is very likely that the approximated value-function converges towards a good solution. The experiments in this dissertation have confirmed these findings. Moreover, a priori knowledge regarding undesirable states can be included in the learning process by defining such undesirable states and evaluating explorations not only concerning the target state but concerning these undesirable states as well.

Because of their aggressive learning behavior and local competence in approximation, networks of radial basis functions have turned out to be a suitable choice for the approximation of value-functions in EBC. Additionally, their behavior is transparent and hence easy to grasp by a human. Multi layer perceptrons are an accurate means for value-function approximation in EBC, too. In contrast to networks of radial basis functions multi layer perceptrons tend to interpolate and extrapolate for unknown regions. They should only be used if a huge amount of training data is available.

In section 4.3, we provide various exploration policies which were designed for different kinds of problems: The backward exploration has been shown to work in cases where no conventional exploration policy manages to find a path to the target state. In case we already know a number of actions (or policies) that lead to the target state, the fixed-action exploration will help us to find out which is the appropriate action (or policy) for a certain region of the state space to reach the target as fast as possible. Supervised exploration has been shown to be very powerful to teach a machine particular behaviors. Any level of initial knowledge can be integrated in the process of exploration by combining the proposed exploration strategies.

The majority of machine control tasks in technical applications are deterministic or at least they possess a unimodal probability distribution with a small variance. As a consequence, it is possible to use a simple projection-function instead of a complex MDP-model, which itself was initially designed for discrete state spaces. In our approach, we work directly in a continuous state space and perform a number of explorations before we exploit the data. This is the main reason why our approach is robust against the incremental summation of noise and other errors that are common in conventional reinforcement learning.

8.2 Future Work

Although we have presented, discussed, and tested a set of algorithms for experience-based learning, a number of questions remain to be addressed in the future.

Future Improvements

Since the proposed EBC algorithms work non-incrementally it would be advantageous to know how many exploration runs are needed until the control system has reached a specified quality target. Ideally, we could use a formal estimation to compute in advance how much training data is needed to reach a certain level of controller quality. In reinforcement learning, a similar question is to what extent we want to exploit gained information in order to explore the state space. If we compare EBC with Q-learning we can see that in Q-learning – in theory – the optimal solution is found. But in practice it may take a very long time to find it, especially if there is a huge number of discrete states which is the case in high-dimensional state spaces.

In chapter 4 we introduced a number of different exploration techniques. In addition, it would be interesting to construct a formal definition of a “reasonable” policy for exploration. At this point, we have named a number of properties that

such a policy should have but we cannot determine in advance what a good policy is. The same point holds true for the majority of reinforcement learning algorithms.

If we do not approximate the policy-function directly we use the value-function or the Q-function. In the latter two cases we can propose an action and estimate the value of the successor state. As long as the set of actions is limited or the action space is small this leads to good results. But if we have to choose an action from a high-dimensional action space it may take a lot of time to find a good action and moreover we cannot guarantee that the chosen action is optimal. Here, we have to interpolate between different actions, perform local searches, or employ other heuristics to obtain an appropriate solution. This problem is already known from conventional reinforcement learning. Future research efforts might focus on this area to further improve the situation.

Outlook on Future Systems

Recapitulatory, we have described a set of methods that supports the design and development of autonomous intelligent systems. The ultimate goal for future systems is that machines autonomously solve complex control tasks with a user only specifying the task. However, present machine learning algorithms do not met this goal yet. Nevertheless, in this dissertation, we provide highly practicable methods that already work in a largely automated fashion. One can see this as a major step towards fully automated intelligent systems of the future. If the open improvements specified above are satisfactorily solved the entire learning process can be automated. In practice, the experience-based learning methods of this dissertation contribute to the implementation of intelligent autonomous machines in industrial applications as well as in our daily life.

Appendix A

Methods of Function Approximation

A.1 CMACs

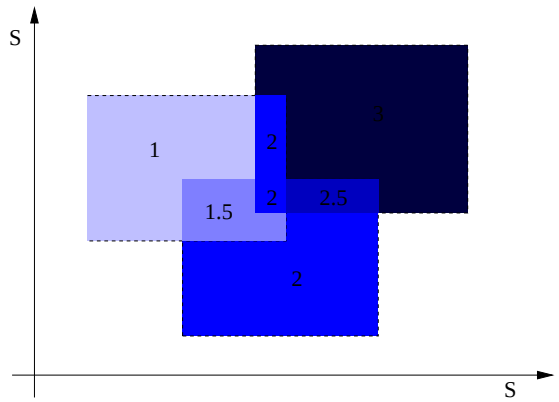


Figure A.1: CMAC with three receptive fields that cover parts of the input space $\mathcal{S}$. Each receptive field returns a certain value. In the overlappings of the receptive fields mean values of the receptive fields involved are returned.

CMACs (Cerebellar Model Articulation Controllers) [Albus 1975a, Albus 1975b, Albus 1981] represent functions that map from multi-dimensional input space to a one-dimensional target space (using one CMAC per output dimension they can map to multi-dimensional spaces as well). They consist of a number of receptive fields that are responsible for computing the output of certain regions of the input space. In general the single receptive fields do overlap and in regions of overlappings the computed output is the mean value of the output of all receptive

fields involved (see fig. A.1). This means that any output of a CMAC is element of $[o_{min}, o_{max}]$, where o_{min} (o_{max}) is the minimal (maximal) output of any receptive field.

Training a CMAC Before training a CMAC the output values of the receptive fields should be initialized (by random for example). Assuming we want train a CMAC using patterns $\langle x, y \rangle, (x \in \mathbb{R}^d$, d is the dimension of the input space, $y \in \mathbb{R})$ we need to compare the current output of the CMAC to x, $CMAC(x)$ with the output wanted, y. Afterwards the output values o_i of all receptive fields i that are responsible for x are adapted according to the following rule:

$$o_i = o_i + \alpha \cdot (y - CMAC(x)) \tag{A.1}$$

where $\alpha \in (0, 1]$ is the learning rate. Using $\alpha = 1$ means to learn aggressively by forgetting all previous data while a small α stands for learning slowly.

Number and Size of Receptive Fields A big number of receptive fields enables a CMAC to learn more complex functions but, however, more training data is needed. Small sized receptive fields allow for great discontinuities in the represented function while large receptive fields and many overlappings help to represent smoother functions.

Value-Function Approximation with CMACs CMACs are popular and often used means for value-function approximation in reinforcement learning. For example, [Kleiner *et al.* 2002] have successfully learned the behavior of soccer robots.

A.2 Multi Layer Perceptrons

Multi Layer Perceptrons (MLPs) are the most popular representatives of neural networks [Hecht-Nielsen 1990, Hertz *et al.* 1991, Bishop 1995, Rojas 1996]. They are based on the ideas of the early model of the perceptron [Rosenblatt 1957, Rosenblatt 1958, Rosenblatt 1962] and represent multi-dimensional functions whose free parameters are tuned by learning algorithms according to the problem at hand. The parameters include the network's *weights*. As an example a single neuron s with its input x_1 and a weight w_1 represents a linear function:

$$s = w_1 x_1 - \Theta \tag{A.2}$$

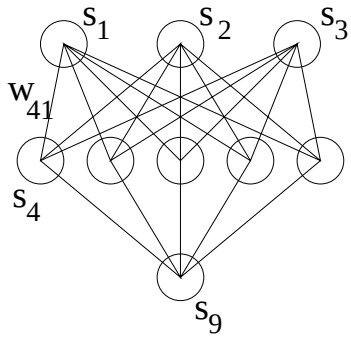


Figure A.2: A MLP-network with a 3-5-1-architecture.

Θ is called the *threshold*. Weights and thresholds are adapted by learning. Therefore training patterns $\langle X, Y \rangle$ ($X \in I\!R^d, Y \in I\!R^n$) are used to specify on which input $X = (x_1, .., x_d)$ the network should return which output $Y = (y_1, .., y_n)$. To approximate nonlinear functions several layers of neurons can be used. In figure A.2 a network with a 3-5-1-architecture is depicted. This network maps from a three-dimensional input space to a one-dimensional output space using five neurons in one *hidden layer*. For instance, the value of s_4 is computed by

$$s_4 = f(\sum_{j=1}^{3} w_{4j} s_j - \Theta_4) \tag{A.3}$$

where f is an activation-function mapping a real number to $[0, 1]$. In most cases the sigmoid function is used for representing f:

$$f(x) = f_{sig}(x) = \frac{1}{1 + e^{-\beta x}} \qquad \frac{\partial f_{sig}(x)}{\partial x} = f_{sig}(x) \cdot (1 - f_{sig}(x)) \tag{A.4}$$

In general the value of any neuron can be computed by

$$s_i = f_{sig}(\sum_{j \in pred(i)} w_{ij} s_j - \Theta_i) \tag{A.5}$$

where $pred(i)$ is the numbers of neurons of the predecessor layer of neuron i.

Gradient Descent The most popular and well known method to adapt the network's weights is the gradient descent algorithm **Backpropagation** [Rumelhart and McClelland 1986]. The error of a network can be computed by

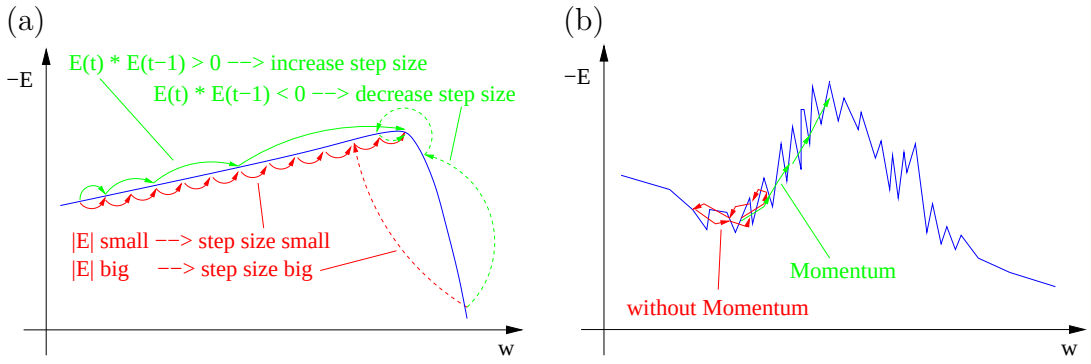


Figure A.3: Subfigure (a): Standard backpropagation uses a small step-size in case of a flat error-function and a big step-size in case of a steep error-function. RPROP works the other way around. Subfigure (b): Momentum term avoids zigzag-movements in rugged error-functions.

$$E = \frac{1}{2}\sum_{k=1}^{n}(s_k - y_k)^2 \tag{A.6}$$

where s_k are the outputs of the network and y_k are the outputs wanted (obtained from the training patterns). From the weights of the network, w_{ij}, a part of the gradient is subtracted to reach a minimum of the error-function E:

$$w_{ij}(t+1) = w_{ij}(t) - \alpha\frac{\partial E}{\partial w_{ij}}(t) \qquad -\frac{\partial E}{\partial w_{ij}} = -\frac{\partial E}{\partial s_i}\cdot\frac{\partial s_i}{\partial w_{ij}} \tag{A.7}$$

where α is a learning rate (which usually is less than 1). Details on Backpropagation can be found in [Hecht-Nielsen 1990, Hertz *et al.* 1991, Bishop 1995, Rojas 1996].

Problems of Backpropagation

Regarding the standard backpropagation algorithm the question remains how to set the learning rate α for the gradient descent. As in equation (A.7) the step-size of the gradient descent is proportional to the absolute value of the gradient. This yields big steps for steep gradients and small steps for flat gradients. According to figure A.3(a) the relation should be the other way around. Further we want to avoid zigzag-movements in the adaptation of the parameters that might occur at rugged passages of the error-function (see figure A.3(b)).

Modifications of Backpropagation

Momentum This method includes a moment of inertia λ (called momentum) in the computation of the new weights:

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \frac{\partial E}{\partial w_{ij}}(t) + \lambda \cdot (w_{ij}(t) - w_{ij}(t-1)) \tag{A.8}$$

The momentum makes the adaptation more smooth. As a result, the optimization process gets not stuck in rugged passages of the error-function that easy.

RPROP The idea behind RPROP [Riedmiller and Braun 1993] is that only the sign of the gradient but not its absolute value has influence on the adaptation of the parameters. If the current gradient and the gradient of the previous learning step have the same sign the step-size is increased, if not it is decreased:

$$w_{ij}(t+1) = \begin{cases} w_{ij}(t) - d_{ij}(t) & if \quad \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ w_{ij}(t) & if \quad \frac{\partial E}{\partial w_{ij}}(t) = 0 \\ w_{ij}(t) + d_{ij}(t) & if \quad \frac{\partial E}{\partial w_{ij}}(t) < 0 \end{cases} \tag{A.9}$$

$d_{ij}(t)$ tells us by what absolute value the weights are changed:

$$d_{ij}(t) = \begin{cases} \kappa^+ \cdot d_{ij}(t-1) & if \quad \frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) > 0 \\ d_{ij}(t-1) & if \quad \frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) = 0 \\ \kappa^- \cdot d_{ij}(t-1) & if \quad \frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) < 0 \end{cases} \tag{A.10}$$

κ^+ and κ^- are element of $(1, \infty)$ and $(0, 1)$, respectively. Experiments have shown that in general $\kappa^+ = 1.2$ and $\kappa^- = 0.5$ work well.

Other algorithms that use an adaptive computation of the step-size include SuperSAB [Tollenaere 1990], Quickprop [Fahlmann 1988], and Delta-Bar-Delta [Jacobs 1988].

Efficient Training of Multi Layer Perceptrons

Learning by Pattern, Block, and Epoch If we use gradient descent algorithms for the adaptation of the parameters of the network there are different possibilities how to do this: We can update the network's parameters after each single computation of a pattern's error. This method is called *learning by pattern* and works well especially for noisy data. If we do not want to update the parameters after computing the gradient for each single pattern we can do it after

a finite number of patterns. This method is called *learning by block* and is computationally less expensive. If we have a finite number of patterns to be trained and want each of them to be virtually learned in parallel we choose *learning by epoch*: We compute the gradients of *all* training patterns and then, in one step, we adapt the parameters by using the sum of all respective gradients. This is called learning by epoch. The advantage, here, is that all patterns are equally considered for learning contrary to learning by pattern (block) where the current pattern (block) has the greatest impact on the parameters and old patterns (blocks) can even be forgotten.

Early Stopping Learning too many epochs can lead to the phenomenon that the error of the training patterns still sinks by every epoch but the ability of generalization (wanted for practical application) gets lost. This phenomenon is called overfitting and forces us to have a look on how our network performs on *validation data* (not used for training) after each epoch of learning. If we learn over a big number of epochs we can choose the network from the epoch with the least error on the validation data for application. Since the validation data are used in order to decide which network is chosen the results of the validation data cannot be presented from a scientific point of view. To see how the chosen network performs on completely unknown data we need a set of *test data* (a third set of data). This approach is called *early stopping* [Sarle 1995].

Weight Decay To keep the weights of a network inside a reasonable interval the idea of *weight decay* was proposed. Experiments showed that this influenced the network's ability of generalization positively [Hinton 1987]. Each learning update forces a decrease of the absolute value of the weights over time:

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \cdot (\frac{\partial E}{\partial w_{ij}}(t) + \xi \cdot w_{ij}(t)) \tag{A.11}$$

The parameter ξ determines how strong the decrease of the weights is.

Optimization of the Topology of Multi Layer Perceptrons

In advance, it is difficult to determine how many layers of neurons and how many neurons are necessary to solve a certain learning problem. Even experienced researchers are sometimes surprised about the complexity of the network that is supposed to be the best one for a certain learning task. Beside human experience there are helpful algorithms that automatize the optimization of the network's topology. These algorithms can be classified into two groups: Constructive and destructive algorithms.

Constructive Algorithms Constructive algorithms build up a network bottom up by iteratively inserting new neurons and connections. Some well known methods are *Upstart* [Frean 1990], *Tiling* [Mezard and Nadal 1989], and *Cascade Correlation* [Fahlman and Lebiere 1989].

Destructive Algorithms Contrary to incrementally building up a network one can start with a big network and remove components that are not relevant. *Pruning*, for instance, removes all connections of weights that have a very small value and thus little impact on the network's performance. *Optimal Brain Damage* [Cun *et al.* 1990] and *Optimal Brain Surgeon* [Hassibi and Stork 1993] both use the Hessian matrix of the weights in order to decide which connections are removed.

A.3 Networks of Radial Basis Functions

RBF-networks were introduced by [Powell 1985, Broomhead and Lowe 1988] and by [Moody and Darken 1989, Poggio and Giorosi 1989]. They employ neurons that consist of radial basis functions. In contrast to multi layer perceptrons (see appendix A.2) the activation of a neuron is not given by the weighted sum of all its inputs but by the computation of a radial basis function. Generally the Gaussian function

$$G_i(x) = e^{-\frac{|x-\mu_i|^2}{2\sigma_i^2}} \tag{A.12}$$

is used, where x is the input of neuron i, μ_i is the basis of neuron i, and σ_i is the amplitude of neuron i.

RBF-Networks are feed forward run and consist of one input layer (x), one hidden layer of Gaussian neurons (G_i), and one output layer (o). The topology of a sample network can be viewed in figure A.4.

The value of an output unit o_k (given a network input x) is computed by

$$o_k = \sum_{i=0}^{g-1} w_{ki} G_i(x) - \theta_k \tag{A.13}$$

where w_{ki} is the weight (=height) of neuron G_i for output o_k and θ_k is a general threshold of output o_k subtracted from the weighted inputs. The Gaussian neurons work as experts for certain areas of the d-dimensional input space. Each neuron's activation depends on its distance to the input vector. In case there is a large quantity of Gaussian neurons, it is sufficient to consider the nearest Gaussian kernels only. Efficient algorithms also used for the nearest neighbor

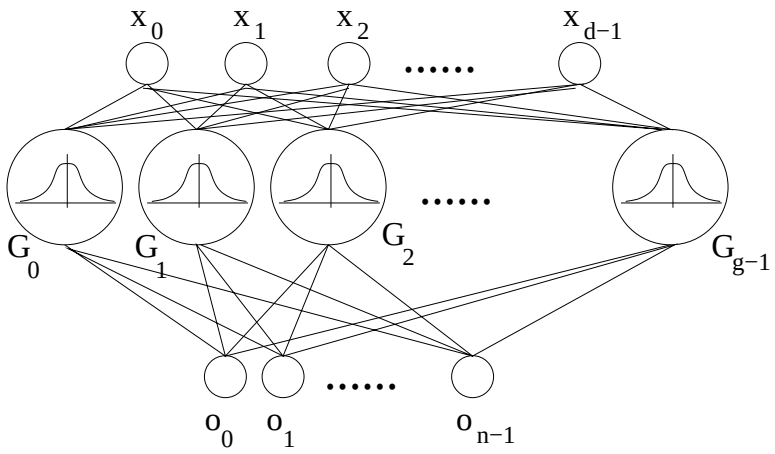


Figure A.4: A sample network with input units $x = \langle x_0 \dots x_{d-1} \rangle$, Gaussian neurons $G_0 \dots G_{g-1}$, and output units $o = \langle o_0 \dots o_{n-1} \rangle$. The neurons of adjacent layers are fully connected.

approach (such as [Arya *et al.* 1994]) can be employed to find relevant Gaussian kernels.

Learning algorithms like Backpropagation and its derivates can be used to adjust the variable parameters of the network:

$$\begin{array}{l} \mu_j, \sigma_j, w_{ki}, \theta_k \\ j \in \{0, .., d-1\}, k \in \{0, .., n-1\}, i \in \{0, .., g-1\} \end{array} \tag{A.14}$$

The following Gradients are used to adapt the network's parameters:

$$\frac{\partial E}{\partial \mu_i} = \sum_{k,p} (o_k^p - y_k^p) \cdot w_{ki} \cdot \frac{x^p - \mu_i}{\sigma_i^2} \cdot e^{-\frac{|x^p - \mu_i|^2}{2\sigma_i^2}} \tag{A.15}$$

$$\frac{\partial E}{\partial \sigma_i} = \sum_{k,p} (o_k^p - y_k^p) \cdot w_{ki} \cdot \frac{|x^p - \mu_i|^2}{2\sigma_i^3} \cdot e^{-\frac{|x^p - \mu_i|^2}{2\sigma_i^2}} \tag{A.16}$$

$$\frac{\partial E}{\partial w_{ki}} = \sum_{p} (o_k^p - y_k^p) \cdot e^{-\frac{|x^p - \mu_i|^2}{2\sigma_i^2}} \tag{A.17}$$

$$\frac{\partial E}{\partial \theta_k} = -\sum_{p} (o_k^p - y_k^p) \tag{A.18}$$

In the above equations, p denotes a specific pattern to be learned while the gradients are computed considering all patterns to be learned. It is even possible

to independently regard the σ_j values for all d dimensions but investigations have shown that there is little if no difference in complexity between a low number of Gaussian neurons with more parameters and a greater number of Gaussian neurons with only one σ_j per neuron. Tools for optimizing the topology are also common in some implementations [Buck 2002]. This means that we dynamically increase or decrease g (the number of Gaussian neurons) while learning (see section 5.8.2 for details).

Efficient Training of RBF-Networks The training is performed similar to training MLPs (see appendix A.2). In addition it is possible to place single Gaussian neurons at strategically reasonable places in the input space. Doing so a priori knowledge can be used.

A.4 Nearest Neighbor Approximation

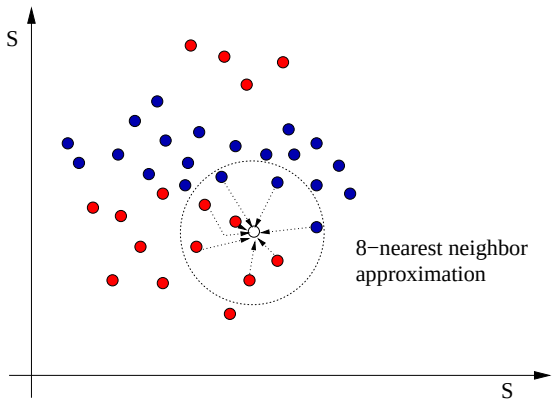


Figure A.5: An example for a k-nearest neighbor approximation. The output is computed depending on the outputs of the k (in this case $k = 8$) nearest inputs.

If we have a large quantity of training data and these data cover the input space reasonably the nearest neighbor approximation is an interesting option for function approximation. The basic idea is simple: Take the k nearest training patterns of an input and compute the output depending on their outputs (see figure A.5). The output o can be computed by simply using the average of the k nearest patterns' outputs (o_i):

$$o = \frac{1}{k} \sum_{i=1}^{k} o_i \tag{A.19}$$

This is useful in cases of huge quantities of training data. [Arya *et al.* 1994], for example, have developed efficient methods to find the k nearest neighbors in high-dimensional spaces. Using the equation above the result is a discontinuous function. To get a more smooth function the influence of each neighbor's output must be dependent on the neighbor's distance:

$$o = \frac{1}{\sum_{i=1}^{k} \frac{1}{1+\delta_i}} \cdot \sum_{i=1}^{k} \frac{o_i}{1+\delta_i} \tag{A.20}$$

δ_i is the distance of neighbor i to the input. Obviously, the drawback of this method is that a metric for the δ_i must be known.

A.5 Decision Trees

A decision tree consists of a number of nodes, leaves, and edges. Each node represents an attribute (see below). The satisfaction of an attribute decides whether one or another edge is chosen to go on towards a leave. Each leave marks the discrete output for a certain subset of the input space.

Definitions

Attributes An attribute of a state is a certain property of the state. It specifies conditions for the state's features. Two examples are: (1) Red can be an attribute if one feature of the state space is the color. (2) The term $\delta \leq 2.0$ can be an attribute if a certain distance δ is a feature of the state space. In general, the satisfaction of attributes splits a space into (at least) two subspaces.

Information To construct a decision tree from training data algorithms rely on the information of an attribute. If we set s^+ the states of the training data that satisfy a particular attribute and set s^- the states that do not, the information I of an attribute is given by

$$I(s^+, s^-) = -\frac{s^+}{s^+ + s^-} \cdot ld\frac{s^+}{s^+ + s^-} - \frac{s^-}{s^+ + s^-} \cdot ld\frac{s^-}{s^+ + s^-} \tag{A.21}$$

Depending on its information, an attribute is located at the beginning of a decision tree or at the end (closer to the leaves). Some common algorithms include ID3, ID5R, and C4.5 [Quinlan 1986].

Appendix B

Robot Soccer

B.1 The RoboCup Challenge

RoboCup is an international research and education program. It focuses on research in AI and intelligent robotics by providing different standard problems related to robot soccer. Robot soccer is a suitable domain for the demonstration of robot skills to the public. But the idea behind this is that methods that can make robots play soccer, can solve other tasks too.

The construction of intelligent autonomous systems, from screw to software, is a big challenge for researchers all over the world. The official motivation of the RoboCup initiative is

By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.

The AGILO RoboCuppers participate in the RoboCup mid-size league. Rules and regulations for the mid-size league consist of three parts:

(1) Official FIFA laws that apply for all RoboCup games.

(2) Additional RoboCup rules that apply for RoboCup games only.

(3) Competition rules that are effective for a certain tournament only.

More informations about RoboCup and the effective rules can be obtained from *http://www.robocup.org*

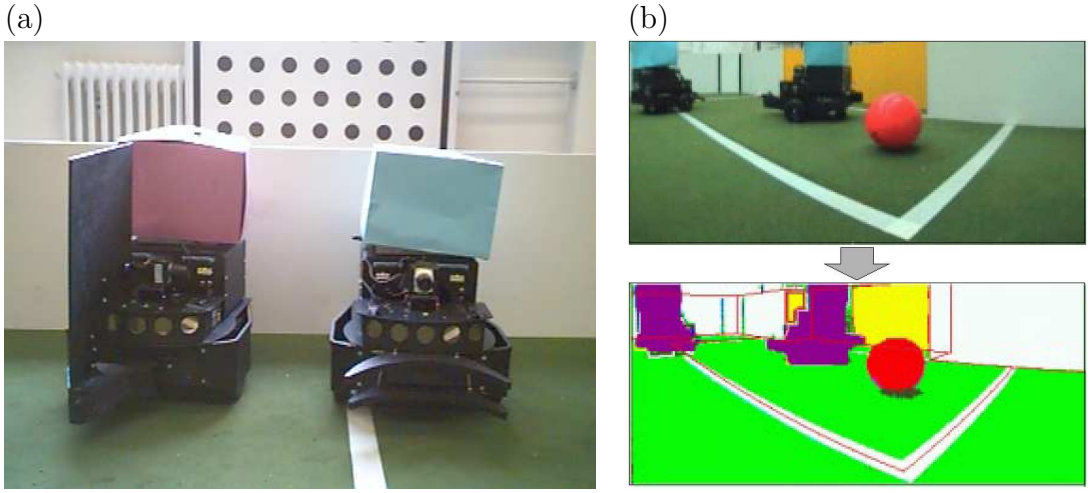


Figure B.1: Subfigure (a): Theodo (goal keeper) and Hugibert (attacker) in 2001. Subfigure (b): What the robots perceive of the world around them.

B.2 The AGILO RoboCuppers

B.2.1 Introduction

The purpose behind the AGILO RoboCuppers (*http://www9.in.tum.de/agilo*) is to provide a reliable platform for (1) pursuing research in multiple fields such as game state estimation, multi-robot cooperation, experience-based learning, and plan-based control, (2) supporting undergraduate and graduate education in computer vision, artificial intelligence, and robotics, and (3) performing software engineering in highly complex software systems. The robot team is described in [Buck *et al.* 2000, Schmitt *et al.* 2001b, Beetz *et al.* 2002] in detail.

B.2.2 Hardware Architecture

The platform of the AGILO RoboCup team is realized using inexpensive and standard hardware components and a standard software environment based on C++.

The robot soccer team consists of four Pioneer I robots [Pioneer 1998] each equipped with an onboard Siemens Lifebook (Pentium 900 MHz CPU, 256 MB RAM, 8 GB hard disk, USB, and Firewire) running under Linux. From 1998 to 2001 an onboard Linux PC (Pentium 200 MHz CPU, 64 MB) was used. The computers are supported by an additional PC outside the playing field used to fuse observations made by the individual robots and to monitor the robots' current states. All the Lifebooks and the PC are linked via a 10 Mbps wireless ethernet (5.8 GHz) [DeltaNetwork, RadioLAN].

Fig. B.1(a) shows two of our Pioneer I robots in 2001. All attackers and defenders have identical shape and equipment except the goalkeeper. The wheels of the robot are controlled by a self-designed controller-board (used for reading the wheel encoders too) that communicates with the Lifebook via the RS232 serial port. Until 2001 we used the original controller-board of the Pioneer I robot for controlling the motors and reading the wheel encoders and the seven ultrasonic sonars.

A color CCD camera is mounted on top of the robot and linked to the Lifebook via Firewire. For a better guidance of the ball we have mounted a simple concave-shaped bar in front of each robot. A custom made kicking device enables the robot to kick the ball in direction of the robot's current orientation.

B.2.3 Fundamental Software Concepts

The software architecture of our system is based on several concurrent modules [Klupsch 1998]. The modules are organized hierarchically into main, intermediate, and basic modules while high-level modules employ modules of lower levels. The main modules are image (sensor) analysis, information fusion, action selection, path planning, and robot control. Beside the main modules the system uses auxiliary modules for monitoring the robots' current states. The key software methods employed by the AGILO RoboCuppers software are:

1. vision-based cooperative state estimation for dynamic environments,
2. synergetic coupling of programming and experience-based learning for movement control and situated action selection, and
3. plan-based control of robot teams using structured reactive controllers (SRCs).

Self Localization, Object Tracking and Data Fusion

The vision module is an essential part of our system. Given a raw video stream, the module has to estimate the state of the robot, the ball, and the opponent robots. Low-level image processing operations are performed using the image processing library HALCON [HALCON].

The AGILO RoboCuppers employ a probabilistic vision-based state estimation method for individual autonomous robots. This method enables a team of mobile robots to estimate their positions in a known environment and track the positions of autonomously moving other objects. All positions of the state estimation module contain a covariance matrix for the description of their uncertainty. The state estimators of different robots cooperate to increase the accuracy and reliability

of the estimations. Cooperation also enables the robots to track temporarily occluded objects and to faster recover their positions after they have lost track of them. A detailed description of the self localization algorithm can be found in [Hanek and Schmitt 2000] and the algorithms used for cooperative multi-object tracking are explained in [Reid 1979, Schmitt *et al.* 2001a].

Experience-Based Learning

The software modules related to experience-based learning have been described in detail in the previous chapters. Videos that show the behavior of the robots can be viewed at *http://www9.in.tum.de/agilo/agilo_videos.html*.

Plan-based Control

Beside experience-based learning the AGILO RoboCuppers employ a robot soccer playbook which is a library of plan schemata that contain how to perform individual team plays. The plays are triggered by opportunities, for example, if the opponent team leaves one side open. The plays themselves specify highly reactive, conditional, and properly synchronized behavior for the individual players of the team. This high-level controller is realized as a structured reactive controller (SRC) [Beetz 2001] and implemented in an extended RPL plan language [McDermott 1991].

B.3 Competitions

The AGILO RoboCuppers conducted RoboCup related research since 1998 and participated in all mid-size league World-Cup-tournaments from 1998 and in three German competitions. The techniques described in this work are part of the AGILO RoboCuppers' software since the year 2000. Both, in the years 2000 and 2001, the AGILO RoboCuppers entered the quarter final. At the RoboCup 2002 in Fukuoka the team participated in the technical challenge competition only. This competition includes one path planning task and one task related to robot cooperation. In this competition the AGILO RoboCuppers took the third place out of 16.

However, the ranking of a team of robots depends on many factors beside the quality of its software. Especially the hardware components of the team have a great impact. In contrast to other teams, the AGILO RoboCuppers use cameras with a view-angle of 45 degrees (to both sides) only and have some of the slowest robots in the mid-size league. For these reasons the results achieved are even more encouraging.

Bibliography

[Alami and Botelho 2002] R. Alami and S.S.C. Botelho: *Plan-based Multi-robot Cooperation.* In M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack (eds.): Advances in Plan-based Control of Autonomous Robots. Selected Contributions of the Dagstuhl Seminar *Plan-based Control of Robotic Agents*, Lecture Notes in Artificial Intelligence 2466, Springer, 2002.

[Albus 1981] J.S. Albus: *Brains, Behavior, and Robotics*. Byte Books, Peterborough, NH, 1981.

[Albus 1975a] J.S. Albus: *A new approach to manipulator control: The cerebellar model articulation controller (CMAC).* Journal of Dynamic Systems, Measurement and Control, pages 220-227, 1975.

[Albus 1975b] J.S. Albus: *Data storage in the cerebellar model articulation controller*. Journal of Dynamic Systems, Measurement and Control, pages 228-233, 1975.

[Alur *et al.* 1999] R. Alur, J. Esposito, M. Kim, V. Kumar, and I. Lee: *Formal modeling and analysis of hybrid systems: A case study in multirobot coordination*. FM'99: Proceedings of the World Congress on Formal Methods, LNCS 1708, pages 212–232, Springer, 1999.

[Arkin 1998] R.C. Arkin: *Behavior Based Robotics.* MIT Press, 1998.

[Arkin 1989] R.C. Arkin: *Towards the Unification of Navigational Planning and Reactive Control*. AAAI Spring Symposium on Robot Navigation, pages 1-5, 1989.

[Arya *et al.* 1994] S. Arya, D.M. Mount, N. Netanyahu, R. Silverman, and A.Y. Wu: *An optimal algorithm for appropriate nearest neighbor searching in fixed dimensions.* In Proc. of the 5th ACM-SIAM Symposium Discrete Algorithms, pages 573-582, 1994.

[Asada *et al.* 1996] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda: *Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning*. Machine Learning Journal, 23:279-303, 1996.

[Barraquand *et al.* 1992] J. Barraquand, B. Langlois, and J. Latombe: *Numerical potential field techniques for robot path planning.* IEEE Transactions on Systems, Man, Cybernetics, 22(2):224–241, March/April 1992.

[Barto *et al.* 1983] A.G. Barto, R.S. Sutton, C.W. Anderson: *Neuronlike adaptive elements that can solve difficult learning control problems.* IEEE Transactions on Systems, Man, and Cybernetics, SMC-13, 5, 1983.

[Beetz *et al.* 2002] M. Beetz, S. Buck, R. Hanek, T. Schmitt, and B. Radig: *The Agilo Autonomous Robot Soccer Team: Computational Principles, Experiences, and Perspectives.* International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS) 2002.

[Beetz 2001] M. Beetz: *Structured Reactive Controllers.* Journal of Autonomous Agents and Multi-Agent Systems, 4:25-55, 2001.

[Belker and Schulz 2002] T. Belker and D. Schulz: *Local Action Planning for Mobile Robot Collision Avoidance.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Belker and Beetz 2001] T. Belker and M. Beetz: *Learning To Execute Navigation Plans* in F. Baader, G. Brewka and T. Eiter (eds): Lecture Notes in Artificial Intelligence, vol. 2174.

[Bennewitz and Burgard 2000] M. Bennewitz and W. Burgard: *A Probabilistic Method for Planning Collision-free Trajectories of Multiple Mobile Robots.* Proc. of the workshop Service Robotics - Applications and Safety Issues in an Emerging Market at the 14th ECAI, 2000.

[Bertsekas and Tsitsiklis 1996] D. Bertsekas and J. Tsitsiklis: *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[Bertsekas 1987] D. Bertsekas: *Dynamic Programming: Deterministic and stochastic models.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[Bishop 1995] C. Bishop: *Neural Networks for Pattern Recognition.* Oxford Press, 1995.

[Boyan and Moore 1995] J. Boyan and A. Moore: *Generalization in Reinforcement Learning: Safely approximating the value function.* In Tesauro, G., D. S. Touretzky, and T. K. Leen (eds.), Advances in Neural Information Processing Systems 7 (NIPS). MIT Press, 1995.

[Brock and Khatib 1999] O. Brock and O. Khatib: *High-speed navigation using the global dynamic window approach.* Proceedings of the IEEE Intl. Conference on Robotics and Automation, 1999.

[Brooks and Mataric 1993] R.A. Brooks and M.J. Mataric: *Real Robots, Real Learning Problems*, in Robot Learning, Jonathan H. Connell and Sridhar Mahadevan, eds., Kluwer Academic Press, 193-213, 1993.

[Brooks 1992] R.A. Brooks: *Artificial Life and Real Robots*, in F. J. Varela and P. Bourgine, editors, Proceedings of the First European Conference on Artificial Life, pages 3-10, 1992.

[Broomhead and Lowe 1988] D.S. Broomhead and D. Lowe: *Multivariable functional interpolation and adaptive networks.* Complex Systems, 2:321–355, 1988.

[Bruce and Veloso 2002] J. Bruce and M. Veloso: *Real-Time Randomized Path Planning for Robot Navigation.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Buck *et al.* 2002a] S. Buck, F. Stulp, M. Beetz, and T. Schmitt: *Machine Control Using Radial Basis Value Functions and Inverse State Projection.* Proceedings of the IEEE Intl. Conf. on Automation, Robotics, Control, and Vision, 2002.

[Buck *et al.* 2002b] S. Buck, M. Beetz, and T. Schmitt: *Approximating the Value Function for Continuous Space Reinforcement Learning in Robot Control.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Buck *et al.* 2002c] S. Buck, M. Beetz, and T. Schmitt: *M-ROSE: A Multi Robot Simulation Environment for Learning Cooperative Behavior.* In H. Asama, T. Arai, T. Fukuda, and T. Hasegawa (eds.): Distributed Autonomous Robotic Systems 5, Springer Verlag, 2002.

[Buck *et al.* 2002d] S. Buck, T. Schmitt, and M. Beetz: *Reliable Multi Robot Coordination Using Minimal Communication and Neural Prediction.* In M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack (eds.): Advances in Plan-based Control of Autonomous Robots. Selected Contributions of the Dagstuhl Seminar *Plan-based Control of Robotic Agents*, Lecture Notes in Artificial Intelligence 2466, Springer, 2002.

[Buck 2002] S. Buck: *RBF++. A C++ Library for Function Approximation Using Networks of Radial Basis Functions.* Manual, Munich University of Technology, March 2002.

[Buck *et al.* 2001a] S. Buck, U. Weber, M. Beetz, and T. Schmitt: *Multi Robot Path Planning for Dynamic Environments: A Case Study.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2001.

[Buck *et al.* 2001b] S. Buck, M. Beetz, and T. Schmitt: *Planning and Executing Joint Navigation Tasks in Autonomous Robot Soccer*. 5th International Workshop on RoboCup, Lecture Notes in Artificial Intelligence, Springer Verlag, 2001.

[Buck and Riedmiller 2000] S. Buck and M. Riedmiller: *Learning Situation Dependent Success Rates Of Actions In A RoboCup Scenario*. Proceedings of the Pacific Rim International Conference on Artificial Intelligence, Lecture Notes in Artificial Intelligence, Springer 2000.

[Buck *et al.* 2000] S. Buck, R. Hanek, M. Klupsch, and T. Schmitt: *Agilo RoboCuppers: RoboCup Team Description*. RoboCup 2000: Robot Soccer World Cup IV, Springer, 2000.

[Buckley 1989] S.J. Buckley: *Fast motion planning for multiple moving objects*. Proc. of the IEEE ICRA 1989.

[Burgard *et al.* 2000] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun: *Collaborative multi-robot localization*. Proceedings of the IEEE International Conference on Robotics and Automation, 2000.

[Burkhard *et al.* 2002] H.D. Burkhard, J. Bach, R. Berger, B. Brunswieck, and M. Gollin: *Mental Models for Robot Control*. In M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack (eds.): Advances in Plan-based Control of Autonomous Robots. Selected Contributions of the Dagstuhl Seminar *Plan-based Control of Robotic Agents*, Lecture Notes in Artificial Intelligence 2466, Springer, 2002.

[Cao *et al.* 1997] Y.U. Cao, A.S. Fukunaga, and A.B. Khang: *Cooperative mobile robotics: Antecedents and directions*. Autonomous Robots, 4, 1997.

[Castelpietra *et al.* 2000] C. Castelpietra, L. Iocchi, D. Nardi, M. Piaggio, A. Scalzo, A. Sgorbissa: *Coordination among Heterogenous Robotic Soccer Players*. Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2000.

[Chen and Luh 1994] Q. Chen and J.Y.S. Luh: *Coordination and control of a group of small mobile robots*. Proceedings of the IEEE International Conference on Robotics and Automation, pages 2315-2320, 1994.

[Cox 1992] E. Cox: *Fuzzy Fundamentals*. IEEE spectrum, October 1992, 58:61, 1992.

[Cun *et al.* 1990] Y.L. Cun, J.S. Denker, and S.A. Solla: *Optimal Brain Damage*. In D.S. Touretzky (ed.), Advances in Neural Information Processing Systems 2, pages 598-605, Morgan Kaufmann, San Mateo, CA, 1990.

[DeltaNetwork] Delta Network Software GmbH, http://www.delta-network-systems.com.

[Dorigo and Colombetti 1994] M. Dorigo and M. Colombetti: *Robot Shaping: Developing autonomous agents through learning.* Artificial Intelligence, 71(2) pp. 321-370, 1994.

[Doya 2000] K. Doya: *Reinforcement Learning In Continuous Time and Space.* Neural Computation, 12, pages 219-245, 2000.

[Doyle *et al.* 1992] J.C. Doyle, B.A. Grancis, and A.R. Tannenbaum: *Feedback Control Theory.* MacMillan, 1992.

[Driankov *et al.* 1993] D. Driankov, H. Hellendoorn, and M. Reinfrank: *An Introduction To Fuzzy Control.* Springer, 1993.

[Duda and Hart 1973] R.O. Duda and P.E. Hart: *Pattern classification and scene analysis.* John Wiley and Sons, New York, London, Sydney, Toronto, 1973.

[Dudek and Jenkin 2000] G. Dudek and M. Jenkin: *Computational Principles of Mobile Robots.* Cambridge University Press, 2000.

[Dudek *et al.* 1996] G. Dudek, M. Jenkin, E.E. Milios, and D. Wilkes: *A taxonomy for multi-agent robotics.* Autonomous Robots, 3(4), 1996.

[Fahlman and Lebiere 1989] S. Fahlman and C. Lebiere: *The cascade-correlation learning architecture.* In Touretzky, D., Advances in Neural Information Processing Systems, 2, 524-532, San Mateo, CA. Morgan Kaufmann, 1989.

[Fahlmann 1988] S. Fahlmann: *An empirical study of learning speed in back-propagation networks.* Technical report, Neuroprose, 1988.

[Forbes and Andre 2000] J. Forbes and D. Andre: *Real-time reinforcement learning in continuous domains.* AAAI Spring Symposium on Real-Time Autonomous Systems. 2000.

[Forbes 2000] J. Forbes: *Reinforcement Learning for Autonomous Vehicles.* Dissertation, University of California at Berkley, Spring 2000.

[Fox *et al.* 1997] D. Fox, W. Burgard, and S. Thrun: *The dynamic window approach to collision avoidance.* IEEE Robotics and Automation Magazine, 4(1), 1997.

[Franklin *et al.* 1994] G.F. Franklin, J.D. Powell, and A. Emami-Naeni: *Feedback Control of Dynamic Systems.* Addison- Wesley, 1994.

[Frean 1990] M. Frean: *The upstart algorithm: A method for constructing and training feedforward neural networks.* Neural Computation, 2, 198-209, 1990.

[Garland and Alterman 1996] A. Garland and R. Alterman: *Multiagent Learning through Collective Memory.* AAAI Spring Symposium Series 1996: Adaption, Co-evolution and Learning in Multiagent Systems, 1996.

[Geibel 2001] P. Geibel: *Reinforcement Learning with Bounded Risk.* In: C. E. Brodley, and A. P. Danyluk, editors, Machine Learning - Proceedings of the Eighteenth International Conference (ICML), pp. 162-169. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[Gerkey and Mataric 2002] B. Gerkey and M.J. Mataric: *Multi-Robot Task Allocation: Analyzing the Complexity and Optimality of Key Architectures.* Technical Report CRES-02-005, Center for Robotics and Embedded Systems, School of Engineering, University of Southern California, September 2002.

[Graham and Ollero 1996] B. Graham and A. Ollero: *An Introduction to Fuzzy Control.* Springer, 1996.

[HALCON] HALCON, *Software Solution for Machine Vision Applications.* http://www.mvtec.com.

[Hanek and Schmitt 2000] R. Hanek and T. Schmitt: *Vision-Based Localization and Data Fusion in a System of Cooperating Mobile Robots.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, pages 1199-1204, 2000.

[Hart *et al.* 1968] P.E. Hart, N.J. Nilsson, and B. Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics, 4(2): 100-107, 1968.

[Hassibi and Stork 1993] B. Hassibi and D.G. Stork: *Second derivates for network pruning: Optimal Brain Surgeon.* In S.J. Hanson, J.D. Cowan, C.L. Giles (eds.), Advances in Neural Information Processing Systems 5, pages 164-171, Morgan Kaufmann, San Mateo, CA, 1993.

[Hecht-Nielsen 1990] R. Hecht-Nielsen: *Neurocomputing.* Addison Wesley, 1990.

[Hertz *et al.* 1991] J. Hertz, A. Krogh, and R. G. Palmer: *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.

[Hinton 1987] G.E. Hinton: *Learning translation invariant recognition in a massively parallel network.* In Goos, G. and Hartmanis, J., editors, PARLE:

Parallel Architectures and Languages Europe, pages 1-13, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1987.

[Huber and Grupen 1997] M. Huber and R.A. Grupen: *Learning to Coordinate Controllers - Reinforcement Learning on a Control Basis.* Proc. of the 15th International Joint Conference on Artificial Intelligence, pp. 1366-1371, Nagoya, Japan, 1997.

[Hwang and Ahuja 1992] Y. K. Hwang and H. Ahuja: *A Potential Field Approach to Path Planning.* IEEE Transactions on Robotics and Automation, vol. 8, 1, 23-32, 1992.

[Jacobs 1993] O.N.R. Jacobs: *Introduction to Control Theory.* Oxford Science Publications, 1993.

[Jacobs 1988] R.A. Jacobs: *Increased rate of convergence through learning rate adaptation.* Neural Networks 1, 1988.

[Jäger 2002] M. Jäger: *Cooperating Cleaning Robots.* In H. Asama, T. Arai, T. Fukuda, and T. Hasegawa (eds.): Distributed Autonomous Robotic Systems 5, Springer Verlag, 2002.

[Jakobi *et al.* 1995] N. Jakobi, P. Husbands and I. Harvey: *Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics*, Third European Conference on Artificial Life (ECAL95), pages 704-720, Springer Verlag, 1995.

[Jennings 1995] N. Jennings: *Controlling cooperative problem solving in industrial multi-agent systems using joint intentions.* Artificial Intelligence, 75, 1995.

[Jensen *et al.* 2002] B. Jensen, G. Froidevaux, X. Greppin, A. Lorotte, L. Mayor, M. Meisser, G. Ramel, and R. Siegwart: *The interactive autonomous mobile system RoboX.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Jugel and Sydow 1998] M.L. Jugel and A. Sydow: *Parallelity in High-Level Simulation Architectures*, in Transaction of the Society for Computer Simulation International, Vol. 15, No. 3, pages 101-103, 1998.

[Kaelbling *et al.* 1996] L. Kaelbling, A. Cassandra, and J. Kurien: *Acting under uncertainty: Discrete bayesian models for mobile-robot navigation.* Proceedings of the IEEE/RSJ IROS 1996.

[Kalmar *et al.* 1998] Z. Kalmar, C. Szepesvari, and A. Lorincz: *Module Based Reinforcement Learning for a Real Robot.* Proceedings of the 6th European Workshop on Learning Robots, Lecture Notes in AI. 1998.

[Kant and Zucker 1986] K. Kant and S. Zucker: *Toward efficient trajectory planning: the path-velocity decomposition.* Int. Journal of Robotics Research, 5(3):72–89, Fall 1986.

[Khatib 1986] O. Khatib: *Real-time obstacle avoidance for manipulators and mobile robots.* International Journal of Robotics Research, 5(1):90-98, Spring 1986.

[Khoo and Horswill 2002] A. Khoo and I. Horswill: *An Efficient Coordination Architecture for Autonomous Robot Teams.* Proc. 2002 IEEE Intl Conference on Robotics and Automation, pp. 287-292, 2002.

[Kickert and van Nauta Lemke 1976] W.J.M. Kickert and H.R. van Nauta Lemke: *Application of a fuzzy controller in a warm water plant.* Automatica 12, pp. 301-308, 1976.

[Kitano *et al.* 1997] H. Kitano, M. Tambe, P. Stone, S. Coradesci, H. Matsubara, M. Veloso, I. Noda, E. Osawa, and M. Asada: *The robocup synthetic agents' challenge.* In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1997.

[Kleiner *et al.* 2002] A. Kleiner, M. Dietl, and B. Nebel: *Towards a Life-Long Learning Soccer Agent.* In G.A. Kaminka, P.U. Lima, and R. Rojas (eds.): Proceedings of the 2002 Intl. RoboCup Symposium, Fukuoka, 2002.

[Klupsch 1998] M. Klupsch: *Object-Oriented Representation of Time-Varying Data Sequences in Multiagent Systems.* In N.C. Callaos (editor) International Conference on Information Systems Analysis and Synthesis (ISAS), pages 833-839, 1998.

[Kobialka *et al.* 2000] H-Ul. Kobialka, P. Schoell, and A. Bredenfeld: *Tools for Assessing RoboCup Behavior* RoboCup Workshop, RoboCup-Euro 2000, Amsterdam, May 28th - June 2nd, 2000

[Kohonen 1988] T. Kohonen: *Self-Organization and Associative Memory.* Springer, 1988.

[Kondo and Ito 2002] T. Kondo and K. Ito: *Reinforcement Learning with Adaptive State Space Recruitment Strategy for Real Autonomous Robots.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Konolige 2000] K. Konolige: *A Gradient Method for Realtime Robot Control.* Proc. of the IEEE/RSJ IROS 2000.

[Konolige *et al.* 1997] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti: *The Saphira Architecture: A Design for Autonomy*. Journal of Experimental and Theoretical Artificial Intelligence, 9:215-235, 1997.

[Kube and Zhang 1994] R. Kube and H. Zhang: *Collective Robotics: From Social Insects to Robots*. Adaptive Behaviour, Vol. 2, No. 2, pages 189-218, 1994.

[Kuo 1995] B.C. Kuo: *Automatic Control Systems*. Prentice Hall, 7th edition, 1995.

[Latombe 1991] J.-C. Latombe: *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Lee *et al.* 1999] T. Lee, U. Nehmzow, and R. Hubbold: *Computer Simulation of Learning Experiments with Autonomous Mobile Robots*, Proceedings of TIMR 99, Towards Intelligent Mobile Robots, Bristol, 1999.

[Lee *et al.* 1998] T. Lee, U. Nehmzow, and R. Hubbold: *Mobile Robot Simulation by Means of Acquired Neural Network Models*, European Simulation Multiconference, Manchester 1998.

[Lengyel *et al.* 1990] J. Lengyel, M. Reichert, B. Donald, and D. Greenberg: *Real–time robot motion planning using rasterizing computer graphics hardware*. Proceedings of SIGGRAPH, pages 327–335, August 1990.

[Leroy *et al.* 1999] S. Leroy, J.P. Laumond, and T. Simeon: *Multiple path coordination for mobile robots: A geometric algorithm*. Proceedings of the IJCAI 1999.

[Levesque *et al.* 1990] H.J. Levesque, P.R. Cohen, and J. Nunes: *On acting together*. In Proceedings of the National Conference on Artificial Intelligence, AAAI press, 1990.

[Lin 1992] L.-J. Lin: *Self improving reactive agents based on reinforcement learning, planning and teaching.* Machine Learning 8, pages 293-321, 1992.

[Martinson *et al.* 2002] E. Martinson, A. Stoytchev, and R. Arkin: *Robot Behavioral Selection Using Q-learning.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Mataric 1998a] M.J. Mataric: *Coordination and Learning in Multi-Robot Systems*. IEEE Intelligent Systems, Mar/Apr 1998, 6-8.

[Mataric 1998b] M.J. Mataric: *Using Communication to Reduce Locality in Distributed Multi-Agent Learning*. Journal of Experimental and Theoretical Artificial Intelligence, special issue on Learning in DAI Systems, Gerhard Weiss, ed., 10(3), Jul-Sep, 357-369, 1998.

[Mataric 1997] M.J. Mataric: *Reinforcement learning in the multi-robot domain.* Autonomous Robots 4, pages 73-83, 1997.

[McCallum 1995] A. McCallum: *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, 1995.

[McDermott 1991] D. McDermott: *A reactive plan language.* Research Report YALEEU/DCS/RR-864, Yale University, 1991.

[Mehlhaus and Rausch 1993] U. Mehlhaus and W. Rausch: *Distributed simulation of robot tasks*, ESS'93 European Simulation Symposium, pages 433–438, Delft, Holland, October 1993.

[Mezard and Nadal 1989] M. Mezard and J.P. Nadal: *Learning in feedforward layered networks: the tiling algorithm.* J. Phys. A: Math. and Gen., 22:2191–203, 1989.

[Michalski 1986] R. Michalski: *Understanding the Nature of Learning*, In Michalski, Carbonell, and Mitchell (eds.), Machine Learning - An Artificial Intelligence Approach. Morgan Kaufmann, Los Altos, California, 1986.

[Minguez *et al.* 2002] J. Minguez, L. Montano, and O. Khatib: *Reactive Collision Avoidance for Navigation with Dynamic Constraints.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2002.

[Moody and Darken 1989] J. Moody and C.J. Darken: *Fast learning in networks of locally-tuned processing units*. Neural Computation, 1, 281-294, 1989.

[Moore and Atkeson 1995] A. Moore and C. Atkeson: *The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional Statespaces.* Machine Learning Journal, 21(3):199-233, 1995.

[Moore and Atkeson 1993] A. Moore and C. Atkeson: *Prioritized sweeping: Reinforcement learning with less data and less real time.* Machine Learning, 13:103–130, 1993.

[Munos and Moore 1999] R. Munos and A. Moore: *Barycentric Interpolators for Continuous Space & Time Reinforcement Learning*. Advances in Neural Information Processing Systems 11, May 1999.

[Murphy 2000] R.R. Murphy: *Introduction to AI.* MIT Press, 2000.

[Noda *et al.* 1998] I. Noda, H. Matsubara, K. Hiraki, and I. Frank: *Soccer Server: A Tool for Research on Multiagent Systems*. Applied Artificial Intelligence, 12, 2–3, pages 233–250, 1998.

[Peng and Williams 1993] J. Peng and R.J. Williams: *Efficient learning and planning within the Dyna framework.* Adaptive Behavior, 1(4), 1993.

[Perez-Uribe 1997] A. Perez-Uribe: *Reinforcement Learning Aircraft Autolander* . Logic Systems Laboratory, Swiss Federal Institute of Technology-Lausanne, http://lslwww.epfl.ch/˜aperez/rlaa.html, 1997.

[Perkins and Hayes 1996] S. Perkins and G. Hayes: *Robot Shaping: Principles, methods and architectures.* Proc. of the AISB Workshop on Learning in Robots and Animals, Univ. of Sussex, UK, 1996.

[Pioneer 1998] Pioneer Mobile Robots, Operation Manual, 2nd edition, Active Media, 1998.

[Poggio and Giorosi 1989] T. Poggio and F. Giorosi: *A theory of networks for approximation and learning.* AI Memo: No 1140, MIT AI Laboratory, 1989.

[Pomerleau 1993] D.A. Pomerleau: *Neural Network Perception for Mobile Robot Guidance.* Kluwer Academic Publishers, Boston, MA, 1993.

[Powell 1985] M.J.D. Powell: *Radial basis functions for multivariate interpolation: A review.* Proc. of the conference on algorithms for the approximation of functions and data, 1985.

[Pynadath and Tambe 2002] D.V. Pynadath and M. Tambe: *Multiagent Teamwork: Analyzing the Optimality and Complexity of Key Theories and Models.* International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS) 2002.

[Quinlan 1986] R. Quinlan: *Induction of decision trees*, Machine Learning 1 (1), 1986

[RadioLAN] RadioLAN Inc., http://www.radiolan.com.

[Reid 1979] D. Reid: *An algorithm for tracking multiple targets.* IEEE Transactions on Automatic Control, 24(6):843-854, 1979.

[Riedmiller and Merke 2002] M. Riedmiller and A. Merke: *Using Machine Learning Techniques in Complex Multi-Agent Domains* . In I. Stamatescu, W. Menzel, M. Richter and U. Ratsch (eds.), Perspectives on Adaptivity and Learning, 2002, LNCS, Springer.

[Riedmiller and Merke 2001] M. Riedmiller and A. Merke: *Karlsruhe Brainstormers - a reinforcement learning approach to robotic soccer II.* In 5th International Workshop on RoboCup, Lecture Notes in Artificial Intelligence, 2001, Springer Verlag.

[Riedmiller *et al.* 1999] M. Riedmiller, S. Buck, A. Merke, R. Ehrmann, O. Thate, S. Dilger, A. Sinner, A. Hofmann, and L. Frommberger: *Karlsruhe Brainstormers - Design Principles*. In M. Veloso, E. Pagello, H. Kitano, editors, RoboCup-99: Robot Soccer World Cup III, Lecture Notes in Artificial Intelligence, pages 588-591, Springer Verlag, 1999.

[Riedmiller 1999] M. Riedmiller: *Tools mit n++: paratest und testnet*, ILKD University of Karlsruhe, in German, 1999.

[Riedmiller 1995] M. Riedmiller: *Dokumentation zu n++*, ILKD University of Karlsruhe, in German, 1995.

[Riedmiller and Braun 1993] M. Riedmiller and H. Braun: *A direct adaptive method for faster backpropagation learning: the Rprop algorithm*, Proceedings of the ICNN, San Francisco, 1993.

[Riedmiller 1993] M. Riedmiller: *Controlling an Inverted Pendulum by Neural Plant Identification*, SMC 93 Conference, Le Touquet, 1993.

[Rojas 1996] R. Rojas: *Neural Networks*. Springer Buch Verlag, 1996.

[Rosenblatt 1962] F. Rosenblatt: *Principles of Neurodynamics, Perceptrons, and the Theory of Brain Mechanism*. Spartan Books, 1962.

[Rosenblatt 1958] F. Rosenblatt: *The Perception: A probabilistic model for information storage and organization in the brain*. Psychological Review 65, pages 386-408, 1958.

[Rosenblatt 1957] F. Rosenblatt: *The Perceptron: A perceiving and recognizing automaton*. Report 85-460-1, Project PARA, Cornell Aeronautical Laboratory, Ithaca, New York, 1957.

[Ross 1983] S. Ross: *Introduction to Stochastic Dynamic Programming*. Academic Press, New York, 1983.

[Rumelhart and McClelland 1986] D.E. Rumelhart and J.L. McClelland: *Learning internal representations by error propagation.* In Parallel Distributed Processing 1, pages 318-362, MIT Press, 1986.

[Rummery and Niranjan 1994] G.A. Rummery and M. Niranjan: *On-line Q-learning using connectionist systems.* Technical report CUED/F-INFENG/TR 166, Engineering Department, Cambridge University, 1994.

[Saffiotti *et al.* 2000] A. Saffiotti, N. B. Zumel, and E. H. Ruspini: *Multi-robot Team Coordination using Desirabilities*. Proceedings of the Sixth International Conference on Intelligent Autonomous Systems, 2000.

[Sarle 1995] W.S. Sarle: *Stopped training and other remedies for overfitting.* In Proceedings of the 27th Symposium on Interface, 1995.

[Schaal 1997] S. Schaal: *Learning from demonstration.* In: M.C. Mozer, M. Jordan, and T. Petsche (eds.), Advances in Neural Information Processing Systems 9, pp.1040-1046, Cambridge, MA, MIT Press, 1997.

[Schmitt *et al.* 2001a] T. Schmitt, R. Hanek, S. Buck, and M. Beetz: *Cooperative Probabilistic State Estimation for Vision-based Autonomous Mobile Robots.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2001.

[Schmitt *et al.* 2001b] T. Schmitt, R. Hanek, S. Buck, and M. Beetz: *Agilo RoboCuppers 2001: Utility- and Plan-based Action Selection based on Probabilistically Estimated Game Situations.* In RoboCup International Symposium, Seattle, 2001.

[Schweikard 1992] A. Schweikard: *A simple path search strategy based on calculation of free sections of motions.* Engineering Applications of Artificial Intelligence, 5, 1, 1 - 10, 1992.

[Sen *et al.* 1994] S. Sen, S. Mahendra, and J. Hale: *Learning to Coordinate Without Sharing Information.* Proceedings of the Twelfth National Conference on Artificial Intelligence, pages 426-431, 1994.

[Simmons *et al.* 2000] R.G. Simmons, D. Apfelbaum, W. Burgard, D. Fox, M. Moors, S. Thrun, and Hkan L.S. Younes: *Coordination for multi-robot exploration and mapping.* In Proceedings of the Seventeenth National Conference on Artificial Intelligence, pages 852-858, 2000.

[Simon 1983] H.A. Simon: *Why Should Machines Learn?* In Michalski, Carbonell, and Mitchell (eds.), Machine Learning - An Artificial Intelligence Approach. Morgan Kaufmann, Los Altos, California, 1983.

[Smith *et al.* 1999] W. Smith, V.E. Taylor, and I. Foster: *Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance.* JSSPP, pages 202-219, 1999.

[Smart and Kaelbling 2000] W.D. Smart and L.P. Kaelbling: *Practical Reinforcement Learning in Continuous Spaces.* In Proceedings of the Seventeenth International Conference on Machine Learning, pages 903-910, 2000.

[Smart 2000] W.D. Smart: *Making Reinforcement Learning Work on Real Robots.* PhD Thesis, Department of Computer Science, Brown University, 2002.

[Stone and Sutton 2001] P. Stone and R. Sutton: *Scaling Reinforcement Learning toward RoboCup Soccer.* Eighteenth International Conference on Machine Learning (ICML), 2001.

[Stone 2000] P.Stone: *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer.* MIT Press, 2000.

[Sukthankar *et al.* 1993] R. Sukthankar, D. Pomerleau, and C. Thorpe: *Panacea: An Active Sensor Controller for the ALVINN Autonomous Driving System.* Tech. report CMU-RI-TR-93-09, Robotics Institute, Carnegie Mellon University, April, 1993.

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

[Sutton 1996] R.S. Sutton: *Generalization in reinforcement learning: Successful examples using sparse coarse coding.* Advances in Neural Information Processing Systems 8, pp. 1038-1044, MIT Press, 1996.

[Sutton 1990] R.S. Sutton: *Integrated architectures for learning, planning, and reacting based on approximating dynamic programming.* Proc. of the Seventh International Conference on Machine Learning, pp. 216–224, San Mateo, CA. Morgan Kaufmann, 1990.

[Sutton 1988] R.S. Sutton: *Learning to predict by the method of temporal differences.* Machine Learning 3, pages 9-44, 1988.

[Takahashi and Asada 2000] Y. Takahashi and M. Asada: *Vision-Guided Behavior Acquisition of a Mobile Robot by Multi-Layered Reinforcement Learning.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, pp. 395-402, 2000.

[Takahashi *et al.* 1999] Y. Takahashi, M. Takeda, and M. Asada: *Continuous Valued Q-learning for Vision-Guided Behavior Acquisition.* Proceedings of the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, pp. 255-260, 1999.

[Takeda *et al.* 2001] M. Takeda, T. Nakamura, and T. Ogasawara: *Continuous Valued Q-learning Method Able to Incrementally Refine State Space.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 2001.

[Tesauro 1994] G.J. Tesauro: *TD–gammon, a self-teaching backgammon program, achieves master-level play.* Neural Computation, 6(2):215–219, 1994.

[Tambe and Zhang 1998] M. Tambe and W. Zhang: *Towards flexible teamwork in persistent teams.* Proceedings of the International conference on multi-agent systems (ICMAS), 1998.

[Tews and Wyeth 2000] A. Tews and G. Wyeth: *Thinking as One: Coordination of Multiple Robots by Shared Representations.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, vol. 2, pages 1391-1396, 2000.

[Tews and Wyeth 1999] A. Tews and G. Wyeth: *Multi-Robot Coordination in the Robot Soccer Environment.* Proceedings of the Australian Conference on Robotics and Automation (ACRA '99), March 30 - April 1, Brisbane, pages 90-95, 1999.

[Thrun *et al.* 2000] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Haehnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz: *Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva.* International Journal of Robotics Research, 19(11), pp. 972–999, 2000.

[Thrun *et al.* 1998] S. Thrun, A. Bucken, W. Burgard, D. Fox, T. Frohlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schimdt: *Map learning and high-speed navigation in RHINO* MIT/AAAI Press, Cambridge, MA, 1998.

[Thrun and Schwartz 1993] S. Thrun and A. Schwartz: *Issues in Using Function Approximation for Reinforcement Learning.* In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, Proceedings of the Connectionist Models Summer School, pages 255-263, Hillsdale, NJ, 1993.

[Tollenaere 1990] T. Tollenaere: *SuperSAB: Fast Adaptive Back Propagation with Good Scaling Properties.* Neural Networks 3, 561-573, 1990.

[Tournassoud 1986] P. Tournassoud: *A strategy for obstacle avoidance and its application to multi-robot systems.* Proceedings of the IEEE ICRA 1986, pages 1224-1229.

[Volpe and Khosla 1990] R. Volpe and P. Khosla: *Manipulator Control with Superquadratic Artificial Potential Functions: Theory and Experiments.* IEEE Transactions on Systems, Man, and Cybernetics, vol. 20, 6, 1423-1436, 1990.

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

[Watkins 1989] C.J. Watkins: *Learning from Delayed Rewards.* PhD Thesis, Cambridge University, England, 1989.

[Weigel *et al.* 2002] T. Weigel, J.S. Gutmann, M. Dietl, A. Kleiner, and B. Nebel: *CS Freiburg: Coordinating Robots for Successful Soccer Playing.* IEEE Transactions on Robotics and Automation, 18(5), October 2002.

[Werbos 1990] P. Werbos: *A menu of designs for reinforcement learning over time.* In Neural Networks for Control, W.T. Miller, R.S. Sutton, and P.J. Werbos, editors, pages 67-95, MIT Press, MA, USA, 1990.

[Yasunobo and Mamdani 1985] S. Yasunobo and E.H. Mamdani: *Automatic train operation system by predictive fuzzy control.* In M. Sugeno (ed.), Industrial Applications of Fuzzy Control, North-Holland, pp. 1-18, 1985.

[Young *et al.* 2002] B.J. Young, R.W. Beard, J.M. Kelsey: *Coordinated Control of Multiple Robots using Formation Feedback.* IEEE Transactions on Robotics and Automation, In Review.

Summary of Notation

Symbol	Meaning	First Mentioned on Page
$\mathcal{A}$	action space of the machine to be controlled	6
$\bar{\mathcal{A}}$	sequence of actions	24
a	action of the machine to be controlled	6
$\mathcal{C}$	cost function	104
c	conditions (decision trees)	16
E	error	123
F	force	23
G	Gaussian kernel	137
f	function	11
f_i	feature with index i	82
$\mathcal{I}$	simulation function for sensory data	27
m	mass	23
$\mathcal{O}$	set of obstacles	63
O	obstacle	62
o	output	132
$\mathcal{P}$	projection-function	7
$\mathcal{P}^+$	forward-projection	37
$\mathcal{P}^-$	backward-projection	37
P	probability distribution	117
p	position	23
Q	function evaluating state and action	49
Q^l	function evaluating state and action in a particular layer l	98
$\hat{Q}$	function evaluating state and action over all layers	99
$\mathcal{R}$	set of robots	103

Symbol	Meaning	First Mentioned on Page
$\mathbb{R}$	set of real numbers	11
R	reward	117
r	robot	80
$\mathcal{S}$	state space of the machine to be controlled	6
$\mathcal{S}$	sequence of states	24
s	neuron	132
t	time	7
$\mathcal{V}$	value-function evaluating a state	45
$\mathcal{V}^+$	value-function considering the target state	49
$\mathcal{V}^-$	value-function considering undesirable states	48
V	velocity	23
V_{tr}	translational velocity	29
V_{rot}	rotational velocity	29
w	weight	99
α	angle	62
α	learning rate	118
β	parameter of sigmoid function	133
γ	discount factor	45
Δ	difference	7
Δt	temporal difference	7
$\Delta\zeta$	update function	7
δ	distance (dependent on a specific metric)	44
φ	orientation	29
π	policy mapping from a state to an action	35
Θ	threshold	49
Υ	sensory data vector	27
ζ	state of the machine to be controlled	6

Index