Technical Report

UCAM-CL-TR-209
ISSN 1476-2986

Number 209

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Efficient memory-based learning for robot control

Andrew William Moore

November 1990

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 1990 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

This dissertation is about the application of machine learning to robot control. A system which has no initial model of the robot/world dynamics should be able to construct such a model using data received through its sensors—an approach which is formalized here as the *SAB* (State-Action-Behaviour) control cycle. A method of learning is presented in which all the experiences in the lifetime of the robot are explicitly remembered. The experiences are stored in a manner which permits fast recall of the closest previous experience to any new situation, thus permitting very quick predictions of the effects of proposed actions and, given a goal behaviour, permitting fast generation of a candidate action. The learning can take place in high-dimensional non-linear control spaces with real-valued ranges of variables. Furthermore, the method avoids a number of shortcomings of earlier learning methods in which the controller can become trapped in inadequate performance which does not improve. Also considered is how the system is made resistant to noisy inputs and how it adapts to environmental changes. A well founded mechanism for choosing actions is introduced which solves the experiment/perform dilemma for this domain with adequate computational efficiency, and with fast convergence to the goal behaviour. The dissertation explains in detail how the *SAB* control cycle can be integrated into both low and high complexity tasks. The methods and algorithms are evaluated with numerous experiments using both real and simulated robot domains. The final experiment also illustrates how a compound learning task can be structured into a hierarchy of simple learning tasks.

Acknowledgements

I would like to thank my supervisor William Clocksin for the initial motivation for this work, and for his help and advice. I am very grateful to Mary Lee and Thomas Vogel who provided valuable and detailed comments on many drafts of the dissertation. Thanks are also due to Thomas Clarke and Barney Pell for many useful and inspiring discussions. I am grateful to Chris Atkeson for introducing me to a number of important pieces of related work.

Declaration

I hereby declare that this dissertation is the result of my own work and, unless explicitly stated in the text, contains nothing which is an outcome of work done in collaboration. No part of this dissertation has already been or is currently being submitted for any degree, diploma or other qualification at any other university.

This dissertation is dedicated to my parents.

Contents

Chapter 1

Introduction

The introduction starts with a simple example of a robot control problem. Motivations for learning control are briefly reviewed, and there is a statement of those areas of the subject addressed by this work. Finally, some guidance is given about the structure of the rest of the dissertation.

This dissertation is about robots which can autonomously develop their own models of the world. Figure 1.1 shows a robot looking at its hand. At all times the controller must choose joint torques to cause the perceived position of the hand to behave in a way which helps achieve some task (such as moving towards the cross-hairs). Such a control problem can be solved if the robot knows how the following are related:

- The perceived **State**: the location of the image of the hand, and also its perceived speed and direction.
- The raw **Action**: the signals sent to the motors.
- The perceived **Behaviour**: the change in perceived position and velocity which then occurs.

This relationship is the composition of many other relationships. How does the torque at a joint depend on the signal to the joint? How does the torque affect the angular acceleration of the joint in this particular configuration? How does the configuration affect the perceived hand position?

This is an example of the central problem of low level robotics—the need to compute the relationship between a number of variables related to the robot's state and the environment. Such relations are termed *world models*. They include kinematic models, hand-eye coordination models, dynamic models, and spatial models. The models are difficult to obtain mathematically for reasons described later. A more appealing, and arguably more practical way to obtain them is through learning.

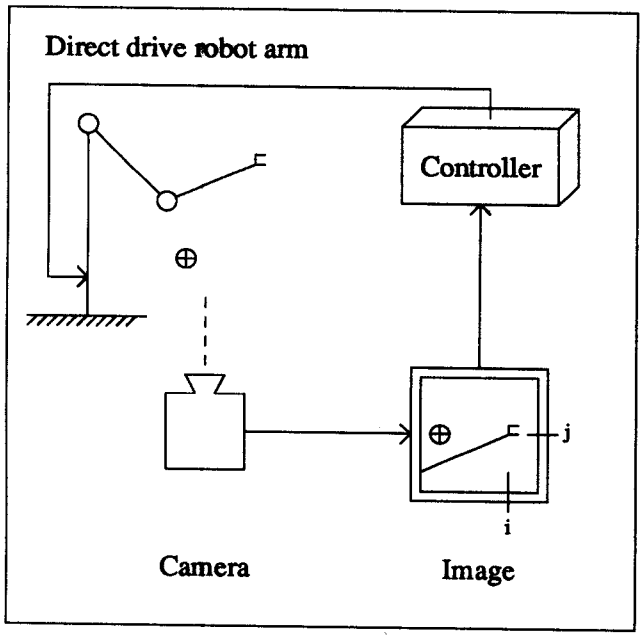

Figure 1.1

A robot looking at its own torque-controlled arm.

1.1 Learning Control: Motivations

A system which can improve itself is an aesthetically pleasing thing. A related motivation for designing learning robots is to understand how learning occurs in biological organisms. A successful automatic learning system might provide an indication as to how animals and people learn. Robot learning seems a particularly good place to begin, because motor learning is perhaps the most basic form of learning behaviour in organisms. It is not, however, guaranteed that engineering a solution provides a complete biological explanation. An analogy is flight, in which the engineered solution differs markedly from the biological solution.

However, the most important motivation is a practical one. Conventional control cannot cope with the sorts of interesting, autonomous machines which would be substantially different from present industrial machinery. The main reason is the difficulty of precoding a sufficiently general world model to accurately take account of all eventualities. The world is complex—even analytic models for simple components such as the relationship between joint angles, velocities, accelerations, and torques are very complex. This is true even when highly idealized and simplified component models are used.

1.2 The SAB Learning System

This dissertation is about a practical, efficient, fast and robust method to obtain robot world models by learning. The use of the word "robot" is for conciseness: the work is also applicable to other dynamic control problems which need multivariate models of the world. The method is called the *SAB Learning System.* The acronym denotes the three components of a dynamic world model: state, action and behaviour. The principal aims of *SAB* learning are listed here:

- **Practical.** The work is strongly motivated by a desire to avoid the "Micro-world" problem. It is concerned with learning in complex high-dimensional state and control spaces with real-valued ranges of variables.
- **Efficient.** The time to update the world model with new knowledge and to use the model that is learned is sufficiently fast that it can realistically occur as the robot operates. This is attained by means of computationally efficient algorithms.
- **Fast.** The learning method is fast so that performance improves very quickly. This is achieved partly by means of a powerful generalization, but primarily by using a *one-shot* learning method: only one presentation of a piece of data is required for its information to be stored. It is not the case that something must be seen a number of times, each time perturbing the world model towards a representation which lessens the error.
- **Robust.** The learning method can cope with disorder in the environment, both in the form of noise, and in the form of either gradual change or sudden unpredictable change. It is also robust with respect to internal parameters, which can be chosen with minimal foreknowledge of the kind of relationship being learned. Finally, it is a method which is hard to get "stuck"—it will not repeat the same error.

This investigation also explains and demonstrates how learning world models can be sufficient to transform the design of robot controllers into simpler design problems. For simple controllers such as trajectory followers, or pick and place, there is almost no additional effort. For compound tasks, the use of learned world models can keep the controller design process at an entirely abstract level which renders the design problem easy for a human, and perhaps even automatable.

1.3 This Dissertation

This dissertation begins with an introduction to the techniques and problems of robot modelling and control, and then an introduction to both earlier and current work in the field of learning control. It formalizes the behaviour of a learned model-based controller and then discusses *how* world models might be represented. It then explains in detail

why the chosen representation can be expected to meet the goals (in turn) of practicality, efficiency, speed, and robustness. The last two features require special attention and are dealt with in their own chapters.

By this time further issues have been uncovered, including the curse of dimensionality, and the search for a useful diversity of experience. These problems are explained and then dealt with using an algorithm called the *SAB Action Chooser*. Following this there is discussion of how best to use the world model to accomplish tasks.

After the main body of the dissertation, a variety of experiments are conducted to evaluate the method's performance. These experiments include

- Learning hand-eye coordination of a real five-jointed arm.
- Learning a visually observed trajectory of a simulated torque-controlled arm under a wide variety of conditions.
- Learning movement control over a wide variety of trajectories for the same arm.
- Learning to juggle.
- Learning to volley a simulated ball into a simulated bucket.

Before the conclusion there is discussion of two additional investigations relating this work to other work: a new method of implementing Albus' CMAC and some experiments with "variable resolution dynamic programming".

Chapter 2

What are Robotic Tasks?

This chapter serves as a simple introduction to some of the issues of robotic control. It begins by introducing and giving examples from the disciplines of (i) robot modelling, (ii) robot control and (iii) robot intelligence. It then discusses the problems of conventional robot modelling and how they affect the higher levels of control.

2.1 Conventional Robot Control

This section provides a brief introduction to the tasks facing the designer of a robot controller. I begin by listing the problems which need to be solved in increasing level of abstractness.

2.1.1 Robot Modelling

Conventionally, modelling is achieved analytically. This is a successful and almost universally applied method in many branches of engineering. A set of primitive axioms which model the behaviour of the primitive components of the physical world are combined using mathematical analysis to model complex systems.

Perception. In order to achieve a task, it is often necessary for objects in the real world to be observed, and from these observations to obtain their real world positions and orientations. An example method of perception is vision, in which the mapping from the image to the real world position and orientation is required.

Kinematics. It is usually necessary for a robot controller to obtain the positions and orientations of particular links and joints in different frames of reference (such as the real world). This computation takes as input (i) the fixed data about the robot, such as its topology and link-lengths, and (ii) a vector $\mathbf{q}$ of current joint positions. A joint position is typically either a *joint angle* if the joint is revolute (as are the joints in Figure 2.1) or else a *joint length* if the joint is prismatic. The output of the computation is the location of the links in world coordinates. Conversely, it is often necessary to take as input a target

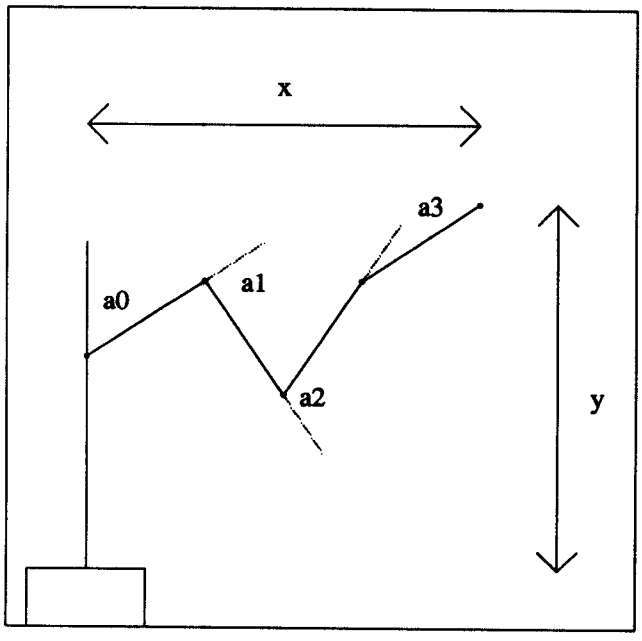

Figure 2.1

A multi-jointed robot manipulator.

position in some other coordinate frame (such as the real world), and to obtain a set of joint positions which would produce this target position. The former computation is of the robot's *kinematics* and the latter is of the robot's *inverse kinematics.*

Dynamics. The robot's dynamic behaviour is determined by the forces which are acting upon it. Some of these forces, such as gravity, are out of its control. Other forces (or torques) are supplied to the robot's joints. The dynamics problem is to compute how the behaviour is affected by the forces acting on the robot.

To formalize this problem, we use the notion of a system's *state.* A *state* representation is a collection of values which contains sufficient information to predict, in principle, the future behaviour of the system, provided the future external and internal forces are also known. A particularly convenient state representation for a robotic manipulator consists of two vectors $\mathbf{q}$ and $\dot{\mathbf{q}}$ which represent the current set of joint positions and their velocities.

The state change, the time derivative of the current state, is determined by the internal and external forces on the arm. Given a current state $\mathbf{s} = (\mathbf{q}, \dot{\mathbf{q}})$, the time derivative of the position component of the state can be calculated from $\mathbf{s}$ trivially—it is the velocity component $\dot{\mathbf{q}}$. The derivative of the velocity, the *joint accelerations* vector $\ddot{\mathbf{q}}$, is the behaviour, dependent on $\mathbf{q}$, $\dot{\mathbf{q}}$ and the *joint torques* $\boldsymbol{\tau}$. This calculation is the dynamics problem. The inverse dynamics problem is the converse. Given a current state $(\mathbf{q}, \dot{\mathbf{q}})$ and a target joint acceleration $\ddot{\mathbf{q}}$, one must compute a set of joint torques, $\boldsymbol{\tau}$, to achieve the target.

2.1.2 Robot Control

Trajectory Tracking. A *trajectory* is a temporal sequence of states

$$((\mathbf{q}_0, \dot{\mathbf{q}}_0), (\mathbf{q}_1, \dot{\mathbf{q}}_1), (\mathbf{q}_2, \dot{\mathbf{q}}_2), \ldots) \tag{2.1}$$

It can be tracked by a sequence of joint accelerations $(\ddot{\mathbf{q}}_0, \ddot{\mathbf{q}}_1, \ddot{\mathbf{q}}_2, \ldots)$ where

$$\ddot{\mathbf{q}}_i = \frac{\dot{\mathbf{q}}_{i+1} - \dot{\mathbf{q}}_i}{h} \tag{2.2}$$

where h is the time step, typically between 1/50th and 1/1000th of a second. $1/h$ is known as the *sampling frequency.* The inverse dynamics model can be used to determine a sequence of joint torque vectors $(\boldsymbol{\tau}_0, \boldsymbol{\tau}_1, \boldsymbol{\tau}_2 \ldots)$ which would cause these ideal accelerations. This method implements open-loop control, and as a result the sequence of torque vectors can be precomputed prior to trajectory execution. For closed-loop control the current state is monitored, and the actual torque vector is modified according to the actual current state.

The advantage of closed-loop control is that, should the behaviour of the manipulator differ from that predicted by the dynamic model, the tracking error can be compensated. There are a variety of reasons that the predicted behaviour is likely to be inaccurate, and these are discussed in Section 2.2.

The compensation can be a function of the error signal. The field of *Control Theory* [Burghes and Graham, 1980] provides a selection of schemes for generating this modification, and also provides the mathematical tools to analyse the stability of the modification strategy. Three common examples are

- **Proportional** (positional) control which adds to the basic precomputed torque a component which is proportional to the current position error, tending to cancel it out.
- **Derivative** (velocity) control which adds a component proportional to the current velocity error. For example, if the required state is stationary, then torques are supplied in the opposite direction to current movement.
- **Integral** control in which the modification varies according to the recent local accumulation of errors.

Different controllers can be combined additively. An example is PD-control in which the chosen torque, $\boldsymbol{\tau}_{\text{now}}$, is defined as

$$\boldsymbol{\tau}_{\text{now}} = \boldsymbol{\tau}_i + K_p(\mathbf{q}_i - \mathbf{q}_{\text{now}}) + K_v(\dot{\mathbf{q}}_i - \dot{\mathbf{q}}_{\text{now}}) \tag{2.3}$$

Such a use of a precomputed inverse model and feedback control is called *feedforward* control. It requires the inverse dynamics model to precompute the necessary torques, as well as general control-theoretic mathematical tools. In particular, the feedback matrices such as K_p and K_v in Equation 2.3 (called *gains*) can be determined analytically. They are

chosen to provide a stable response in a short time. Generally, this mathematical analysis requires the assumption of local linearity in the dynamic model.

Because the error is monitored and reduced, acceptable performance can occur even if the model is only a simple approximation. The extreme case is where no inverse dynamics are computed, and each joint torque is computed entirely according to the current position and velocity errors. Each joint actuator is an independent *servomechanism*: a one-variable control system which continually tries to track the input signal with its output signal by means of linear feedback control. For speeds which are not low, this extreme approach results in large trajectory tracking errors.

Another scheme is *Computed Torque Control* [Fu *et al.*, 1987; An *et al.*, 1988], in which modification can also be based directly on the inverse dynamics model. The ideal acceleration to take us back to the trajectory is computed, and then the torques to achieve this acceleration are computed using the inverse dynamics.

If $(\mathbf{q}_{\text{now}}, \dot{\mathbf{q}}_{\text{now}})$ is the current state, and we are meant to be at the ith state in the trajectory, then we attempt to apply acceleration

$$\ddot{\mathbf{q}}_{\text{now}} = \ddot{\mathbf{q}}_i + K_p(\mathbf{q}_i - \mathbf{q}_{\text{now}}) + K_v(\dot{\mathbf{q}}_i - \dot{\mathbf{q}}_{\text{now}}) \tag{2.4}$$

If the trajectory is controlled in this fashion by accelerations it is a *linear system*, defined in Section 4.2. It can be shown that, provided the gains are not too high, this will converge to an accurate tracking of the trajectory.

Computed torque control is computationally expensive because the inverse dynamics must be computed in real time. The extent to which the model is correct affects the accuracy and stability of the trajectory tracking. At some cost in accuracy a simpler model of the robot dynamics could be used. An example would be a model which only took gravitational forces into account.

In summary, trajectory tracking can be achieved by suitable use of the inverse dynamics model. Three possible methods are

- Open-loop control, using precomputed inverse dynamics. This does not check for errors.
- Closed-loop control using precomputed inverse dynamics and also modifications as a function of the error signal. This requires extra mathematical models and analysis.
- Closed-loop control by dynamically computing the ideal current accelerations and in turn the torques to achieve the accelerations.

Balancing. Typically, remaining in a static position is an easy application of closed loop control, where the error signal is simply the difference between the current state $(\mathbf{q}, \dot{\mathbf{q}})$ and the ideal state $(\mathbf{q}_{\text{ideal}}, \mathbf{0})$. The state is always changed to lessen the error. However, in some dynamic situations, no error-decreasing joint torques may be available. This can occur if any of the joint actuators are insufficiently strong to provide the required torque.

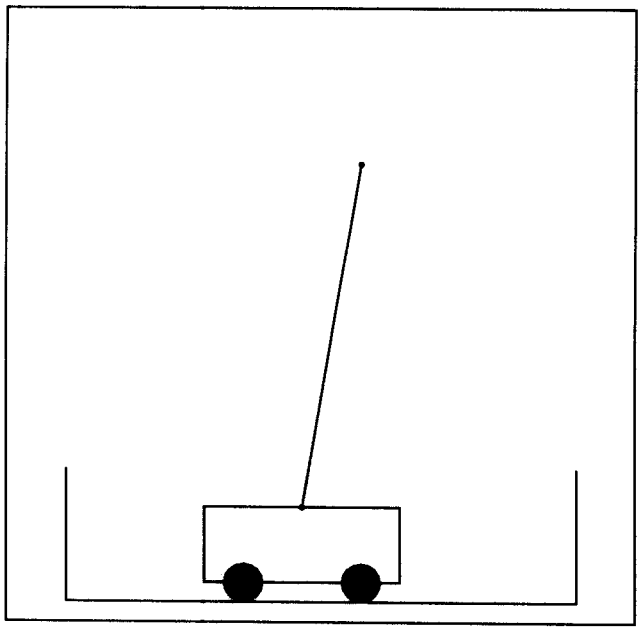

Figure 2.2

The Cart-Pole problem.

In some systems there might even not be an actuator at the joint. In this case, a global strategy for returning to the goal is required: local adjustments are insufficient. This is a stability problem. Perfect knowledge of the kinematics and dynamics would not be sufficient to solve this directly; some intelligence is also needed to develop a strategy for stability.

The classic example is the pole balancing problem, described in [Michie and Chambers, 1968]. This is depicted in Figure 2.2. The cart can be moved left or right along a bounded track. The base of the pole is fixed to the cart by a revolute joint with no actuator. The only control over the angle of the pole is thus indirect, by means of thrusts to the cart. Balance has to be non-local: there are states in which to prevent eventual disaster, the state vector must be moved further from the goal state.

2.1.3 Robot Intelligence

Obstacle Avoidance. To move the manipulator from one static configuration to another, the controller can track a straight line, uniform speed trajectory through joint space. However, during the transition the arm sweeps out a volume of space which must not coincide with any obstacles. Obstacle avoidance involves finding a trajectory in which none of the intermediate configurations cause a collision.

For obstacle avoidance, a specification of the shape of the robot is required, which is defined as a mapping from $\mathbf{q}$, the joint angles, to the space of three-dimensional solids. This *spatial model* is implemented by combining standard three-dimensional geometrical

algorithms with the robot kinematics. A candidate trajectory can be tested by using the spatial model to ensure that no $\mathbf{q}_i$ in the trajectory maps to a solid which intersects with any obstacle. One possible method of solution is to obtain trajectories using a generate-and-test procedure. The search for a valid trajectory can be guided by an evaluation function. This function scores different configurations badly for points near obstacles and well for points far away. The trajectory search follows directions down the gradient of this function.

Autonomy. A highly sophisticated robot controller would require abilities which are at the moment only available to biological systems. These include planning, inter-agent communication and failure management. Such abilities would require a knowledge of the environment which might have to be obtained by learning. The study of these problems are not restricted to robotics, but form much of the general field of Artificial Intelligence.

For example, [Firschein and others, 1986] considers the design of an autonomous robot to assist in the building of a space station. The bandwidth for communication with the robot would be low, unreliable, and with a time delay. As a result, it would not be practical to have a human teleoperating each task, and yet a fixed program specification would not cover the details of the wide range of tasks which the robot would be expected to achieve. Instead, the robot would be given relatively abstract task specifications, and would be left to compute locally the means to achieve them.

2.1.4 Discussion

These were examples of the tasks facing robotic designers. Others include Actuator Modelling, Control Languages and Trajectory Planning and Optimization. The examples were in a roughly increasing order of complexity, and generally the earlier tasks can be used within the solution of later tasks.

2.2 Robot Control: Difficulties

The previous section considered some of the issues addressed by the designer of a robot control system. Now let us explore the difficulties with these conventional solutions. The discussion is split into two sections. The first discusses the problems of robotic mathematical modelling and the second considers the effect that deficiencies in the models have on robot control.

2.2.1 Mathematical Modelling of the Robot's World—the Problems

1. The mathematical model is built from a set of axioms of the physical behaviour of the world (such as Newton's laws of motion). Some low level system components are treated as being atomic, having the ideal properties required to fit these axioms. As a result, the extent to which the model is correct depends on the extent to

which the low level components do in fact match the requirements of the axioms. For example, the joints in the arm may be modelled as two rigid links rotating around a common frictionless axis. If in reality there is friction in the joint then this mismatch between the idealized assumption and the world can lead to general model inaccuracies. If instead the friction is modelled, it might be treated as growing linearly with angular velocity. This too, is only an approximation, and the designer must check experimentally that the inaccuracies are sufficiently small.

2. The mathematical model describes the behaviour in terms of the explicit system variables such as angles and velocities and also a large number of system parameters. The system parameters include features like the mass and moments of inertia of each link, link lengths, coefficients of friction, sensor locations, and the relationship between actuator signals and the torques produced. The values of these parameters cannot be obtained from the theory, and so must be measured. Many of these measurements cannot be made directly and so have to be computed from observations of other features of the system (these computations will in turn be based on mathematical models, with the same set of problems). The accuracy of these parameters affect the model accuracy, often critically.

3. The short term dynamic changes in the system's state are included in the dynamic model. However, it is likely that there will be other changes in the system with time. These include gradual changes in the performance of components due to wear and tear (for example, the joint becomes less stiff). Gradual changes could theoretically be incorporated, but in practice there are no techniques to model such wear. Other changes are the result of unpredictable system perturbations (for example a camera being jogged). These, again, are not possible to model. The system designer must assume that small changes have little impact on the model accuracy, and must regularly inspect the equipment to ensure that no significant changes have occurred.

4. Generally, the differential equations resulting from the models are not analytically soluble and so need to be approximated, or computed numerically, in which case issues of numerical stability arise.

5. The generation of mathematical models evidently requires a great deal of expertise. The designer needs to apply knowledge about the axioms of the physical world behaviour. It is necessary to judge which aspects are important to model and which are unlikely to affect accuracy. Then a large degree of mathematical dexterity is required to combine the components of the model and solve the resulting equations. Some aspects of this system might be automatable using algebraic manipulation software such as REDUCE [Hearn, 1973], but generally it requires the time of expensive, highly skilled experts.

6. The mathematical models require enormous computational power. This greatly increases the expense of calculating real time dynamics and inverse dynamics for tasks such as trajectory tracking. However, increasingly powerful computers will eventually be able to deal with this problem even for complex dynamic systems.

2.2.2 Discussion of Robot Control

The argument in this section will be that the high level robotic problems are always one of the following:

- **Search problems.** These are examples of more general problems, particularly those from the fields of optimization and Artificial Intelligence (AI).
- **Modelling problems.** These are caused by inaccuracies in mathematical modelling.

This argument is supported by considering some of the tasks discussed in the previous section. Trajectory tracking is very poor if it is open-loop and the model is even marginally inaccurate. If control is closed-loop then the error caused by an inaccurate model is reduced, and typically consists of the actual state lagging behind, or never quite reaching, the desired trajectory. A further problem for computed torque or feedforward control for trajectory tracking is the enormous amount of computation required. However, other than these modelling problems, there are few difficulties in designing a trajectory tracker.

Balancing can be achieved by trying to track the static trajectory. It is made harder in the case where the system is in a state of serious imbalance: an attainable trajectory back to safety must be deduced, and there is no entirely general way to obtain such a trajectory analytically from the mathematical model. If the strategy is to be obtained automatically, search techniques must be applied.

Obstacle avoidance relies on a spatial kinematic model. Given a goal and a set of known obstacles then even if the model is entirely correct, it will not be able to prescribe a suitable trajectory. The solution is a search problem in three-dimensional space.

Autonomy depends on an abstract representation of the world and upon many of the general problem solving abilities proposed and investigated in a variety of fields in AI. Abstracting some aspects of the world, such as placing everything in a uniform coordinate system, is a modelling problem. Other aspects of autonomy are from the AI domain.

This dissertation concentrates on a solution to modelling problems. When the designer of the high level control is confronted with a search problem they have two alternatives

1. To perform the search and planning tasks manually, and encapsulate the knowledge in a program.
2. To use the best automatic techniques from the optimization and AI fields.

The conclusion is that for many robotic design tasks, at least one of these alternatives will be entirely adequate. This is because the design tasks may be in domains which are restricted enough to allow such knowledge to be expressed, or for planning to take place in realistic time. The majority of the dissertation will focus instead on how to reliably and accurately learn models of the world.

Chapter 3

Learning Robotic Tasks

This chapter surveys the related literature in learning control. It begins with early work, and then introduces the important distinction between work which learns action maps and work which learns world models. A particularly important aim of the survey is to show that earlier work in the field can usually be regarded as learning some relationship between state, action and behaviour. Other important issues in learning control are also surveyed. Following that, some particularly relevant recent work is summarized. At the end of the chapter is a description of how this dissertation fits in with the issues discussed.

3.1 The Birth of Learning Control

The mechanical governor, invented by James Watt in 1788, is a device which regulates the speed of a rotating shaft. If the rotation increases then a pair of balls move further apart. This movement is transmitted mechanically to the engine producing the rotation, and causes it to work less hard, for example by closing a gas flue. Conversely, if the rotation is less than ideal, the balls are closer and this similarly causes the engine to increase its output. It is an elegant system which controls itself automatically, by sensing the performance and then making adjustments to bring the performance closer to ideal. Such systems have been common in mechanical engineering for over a century. These are called *closed-loop* control systems.

During this century, the use of analogue electronics has allowed a much more flexible approach to such systems. Measurements of the performance can be encoded as electrical signals instead of mechanical positions. From this, more sophisticated forms of closed-loop control have been possible. More recently, the signals have been encoded and processed digitally, allowing yet another increase in the flexibility and amount of the information available.

The large amount of information which can now be obtained about a system leads to the question of how it can best be used. One approach is to use it in the same manner as

earlier generations of controllers. This provides robust but mundane control. An example of this approach are servo-driven robotic manipulators. Each joint is moved by specifying the desired position of the joint, and then this simple closed-loop control is used for each joint independently to continually make adjustments until the position of each joint is as required.

A more interesting possibility is to use more of the information received through the sensors to autonomously improve performance by *learning* about the world. The benefits are summarized here.

- **Optimality.** Control of complex systems such as manipulators can become quicker and more accurate.
- **Self Programming.** Complex systems do not need to be analysed and modelled by expensive human experts. Instead they can discover their own abilities.
- **Model of Intelligence.** A system which improves its own performance provides a possible model for the behaviour of biological systems.
- **Disorder.** A system which monitors its own performance might be able to cope with a noisy, or non-stationary environment.

The difficulty is that there is a very large amount of information which can be used. Many trade-offs are available between (i) the reliance on standard control techniques and (ii) the intensive processing of dynamic information. One trade-off is to use the information to automatically make adjustments to the controller at a high level. This is discussed in the next subsection.

3.1.1 Adapting the Higher Levels of Controllers

The standard closed-loop controller makes adjustments at a low level. If a servo-controlled arm tries to follow a trajectory it continually monitors the current error in the position and velocity. In addition to this, adjustments can be made at a higher level of control. Thus, after an attempted trajectory has been completed, the data collected can be used to adjust, in advance, the torques which will be supplied to each joint at each time step next time the trajectory is attempted. This idea is the basis of **Adaptive Control** [Phan *et al.*, 1990].

A different example of the idea of making adjustments at higher levels of abstraction is **Task-Level Learning** [Aboaf *et al.*, 1989]. An example of this is throwing a ball at a target. Conventional modelling and control are used to throw the ball, but when it fails (due to modelling errors) an adjustment is made to how hard the ball is thrown. If it had fallen short it is thrown harder next time, and if it had overshot it is thrown less hard. The idea was applied successfully to a real bat and ball juggler, the lower levels of which had been built using mathematical models of the system components. With no task-level

learning the ball was usually dropped after only two or three hits. With task level learning the time to failure was substantially increased to typically between eight and twenty hits.

3.1.2 The Basis of Learning Control

The basis for learning control was established during the 1960s and is consolidated in the expository paper [Fu, 1970]. Here, I summarize the paper. Fu cites a variety of possibilities for learning control.

- **Classification from a teacher.** The first possibility was to learn to *classify* states according to which action should best be applied. This drew on the emerging field of **Pattern Classification** [Duda and Hart, 1973]. If a sample of states with known optimal actions are given, then a pattern classifier can be learned which, when given a state not in the original sample, produces an action which, according to some predefined criterion, best agrees with the sample points. Possible criteria included the assumption that the optimal control actions were *linearly separable*, in which case the learning controller tried to find a hyperplane to partition the state space in a manner which agreed with the sample points. This method did not actually use information from the environment to improve performance, but instead relied on a teacher to tell it what was correct.

- **Reinforcement learning.** If, after each action is applied, part of the information from the environment is a signal saying how ideal that action had been, then this signal could be used to improve performance. This is *reinforcement learning.* In this work the state space was partitioned into different regions. Within each region the relative probabilities of attempting alternative actions were modified according to the reinforcement signal. The use of a reinforcement signal increases the autonomy of the learning, but is still a tricky requirement: many systems, while having a global goal, find it hard specify local goals which would lead to the global performance.

- **Stochastic automata.** A stochastic automaton has a finite number of internal states which are traversed between each control cycle. The input to the automaton is one of a finite number of reinforcement signals from the world. The output of the automaton is one of a finite number of actions it applies. The state transition function is a stochastic matrix, which is modified by a scheme designed to increase the probability of causing outputs which produce higher levels of reinforcement.

Fu's overview also mentions some of the issues which, since the date of the paper, have been further developed in the field. These include the problems of learning in a world which can change unpredictably (a *non-stationary* environment), the problems of a slow learning rate and the question of how learning controllers can be linked together. The history of these and other issues are discussed in Section 3.3, but first I will explain how different approaches to learning control developed from these seeds.

3.2 What Should be Learned?

There are two fundamentally different things that a learning control system can attempt to acquire. One is the **Action Map**

$$\textbf{State} \rightarrow \textbf{Action} \tag{3.1}$$

and the other is the **World Model**, which in this disssertation is interpreted as being of the form

$$\textbf{State} \times \textbf{Action} \rightarrow \textbf{Behaviour} \tag{3.2}$$

which is often learned instead as the **Inverse World Model**

$$\textbf{State} \times \textbf{Behaviour} \rightarrow \textbf{Action} \tag{3.3}$$

The inverse form of the world model is dangerous to learn as it may not be a function. Given a current state and a desired behaviour, there might in some contexts be no actions to achieve the behaviour, and in other contexts multiple actions. Successful learning of an inverse world model requires extra domain knowledge that these problems will not occur.

During the remainder of this chapter, all the work will be surveyed using my interpretation of learned mappings as being either action maps or world models that relate state, action and behaviour. The next two subsections describe early examples of each of these. Following that, subsection 3.2.3 contrasts the two approaches.

3.2.1 Michie and Chambers: BOXES

An early, and classic, example of action map learning is the BOXES pole-balancing system [Michie and Chambers, 1968]. The system to be controlled is a pole balanced on a cart. The cart can be thrust left or right at each control cycle, but these are the only permitted actions. The state of the system (which consists of four values: the position and velocity of the cart, and the angle and angular velocity of the pole) can be observed at each control cycle. The goal is to prevent the pole from falling, and this is the only specification which is given to the learning system.

The systems learns the action map:

$$\underbrace{X_{\text{cart}} \times \dot{X}_{\text{cart}} \times \theta_{\text{cart}} \times \dot{\theta}_{\text{cart}}}_{\textbf{State}} \rightarrow \underbrace{\{\texttt{Left}, \texttt{Right}\}}_{\textbf{Action}} \tag{3.4}$$

The representation of the mapping is a four-dimensional array, indexed on the quantized state variables. For example, all x coordinates of the cart between 35ins and 21ins to the left of the cart are considered behaviourally equivalent. Five grades of x position are distinguished. In total there are 225 entries in the array (these entries are the boxes after which the system is named). The contents of the boxes affect what action is chosen should the system's state ever coincide with the box. It contains statistics of previous decisions

made when the box was entered. These identify the mean survival time when the `Left` action was taken and mean survival time when the `Right` action was taken. The decision as to which direction to choose is biased according to which is expected to have longer life. Thus as learning continues, decisions which lead to disaster are gradually eliminated, even if they do not lead to immediate disaster.

This system thus displays the desirable property that a local reinforcement signal is not required—all learning is done simply from the final outcome of the trial. The system did indeed manage to learn to balance the pole within typically a thousand balancing attempts.

3.2.2 Raibert's Parameterized Method

Raibert's work [Raibert, 1978a; Raibert, 1978b] learned to control a real torque driven robot manipulator. It learned the following inverse world model:

$$\underbrace{\text{Joint Angles} \times \text{Joint Velocities}}_{\textbf{State}} \times \underbrace{\text{Joint Acc'ns}}_{\textbf{Behaviour}} \rightarrow \underbrace{\text{Joint Torques}}_{\textbf{Action}} \tag{3.5}$$

This is again represented by a quantized multi-dimensional array. The state space is six-dimensional with ten quantization levels and so there are 10^6 cells (hash coded to save memory). Each cell corresponds to a simple local model of the behaviour of the robot. This learning system uses domain knowledge about the form of equations of motion of a robot arm. Such equations have parameters which vary throughout the state space but can be assumed constant within each box. The values of these parameters can be estimated by, for each cell, recording the real world experiences of the robot and then inverting the known (linear) form of the local model to obtain the parameters. The method of estimating local parameters provides a very accurate model with the disadvantage of needing to assume a certain form for the dynamic equations of motion.

The learning system was given the task of following a prespecified trajectory. This it did by using the learned model to precompute the necessary torques. During execution of the trajectory, no feedback was used. Despite this open-loop control, the performance was good and improved during learning (though it did not reach perfect behaviour). Convergence took approximately 2000 trials.

Raibert carried out some further tests illustrating important features of learning control. Firstly he tested performance in an environment which changed unpredictably over time. The experiment consisted of a period of normal learning after which, unknown to the controller, the dynamic behaviour of the arm was changed by adding a weight to a joint. A second experiment similarly attached a spring to a joint. The arm adapted, but slowly, to the changes.

His second extra experiment was to learn one trajectory and then try executing a nearby trajectory. This was to see how successful the *generalizing* abilities of the learning

system were. The results showed that a trajectory was learned more quickly if a nearby trajectory had been previously learned.

3.2.3 Learning Action Maps or World Models?

Subsequent investigations in learning control have differed as to which of these two approaches is adopted. This survey will distinguish clearly for each piece of work it examines which kind of mapping is learned. This is because it has a large impact on the applicability, expected performance and utility of the learning system concerned. There is a trade-off between the usefulness of what is learned and the expected ease and speed of learning.

Learning action maps provides a more useful end-product because the learned controller knows what to do at each state it is in. The disadvantage is that an action map is generally hard to learn. It is not always clear whether current performance can be improved, and if it can be, how to improve it. A very simple example of this is the two armed bandit problem, described more fully later, which is a system with no state and only two possible actions but for which an optimal solution is still not known.

Despite these serious difficulties, action map learning has been attempted with some success [Michie and Chambers, 1968; Barto *et al.*, 1983; Kaelbling, 1990b; Simons *et al.*, 1982; Gordon and Grefenstette, 1990]. As well as Michie and Chamber's Pole Balancer, [Barto *et al.*, 1983] have implemented a learning controller which balances a pole considerably more quickly with the same delayed reinforcement signal (remember, the pole controller only gets told about its performance when the pole falls). The improvement is by *learning* an immediate reinforcement signal. The reinforcement signal scores an action decision as good if it moves the pole state into a superior state and bad if it moves it into an inferior state. The relative qualities of states are estimated by a record of the expected time to failure starting from each state. The precise definition of this value is hard to pin down. It is trying to estimate the expected time to failure from the current state if the optimal controller were used, but it is estimating this by means of the expected time to failure if the current controller is used. Unfortunately this recursive definition could have multiple solutions and so is not necessarily well-defined. In practice in this and other work (e.g. [Jordan and Jacobs, 1990]), this ambiguity does not seem to cause a problem.

A very recent investigation [Kaelbling, 1990a; Kaelbling, 1990b] thoroughly consolidates work in this area as "Learning in Embedded Systems".

Learning world models is much easier, because it is based on objective observations about the world. If performance is inadequate then it is because the observed behaviour differs from the predicted behaviour. In such a case it is clear how the world model should be updated—it should reduce or eliminate the error. For this reason, model-based learning control systems have been more popular [Raibert, 1978b; Miller, 1989; Mel, 1989; Atkeson, 1989; Zrimec and Mowforth, 1990; Sutton, 1990]. There is a sacrifice to be made for the relative ease of learning: although the world may be modelled, it is not necessarily clear how to use this model. The investigations mentioned above deal with this problem

in a variety of ways.

- **Weak AI or optimization.** In [Christiansen *et al.*, 1990] a controller learns how a flat block behaves when the tray upon which it is lying is tilted by a robot. The experiments involve a real, visually observed, robot. The world model learned is

$$\underbrace{\text{Start pos'n and orientation}}_{\textbf{State}} \times \underbrace{\text{Tilt angles}}_{\textbf{Action}} \rightarrow \underbrace{\text{End pos'n and orientation}}_{\textbf{Behaviour}} \tag{3.6}$$

  The representation is by means of a quantized array. The robot is given a goal position and orientation. The current position and orientation is observed. There is generally not an action which could immediately produce the desired goal, and so instead a standard search is carried out with reference to the learned world model to find a sequence of actions to achieve the goal.

  The tray tilting work explores further learning control issues discussed shortly.

  Other examples of model-based learning which use search and optimzation are [Mel, 1989] which performs a best first search to produce an obstacle-avoiding positional trajectory to reach visual goals and [Sutton, 1990] which uses dynamic programming based on the learned model to plan simple maze paths to a goal.

- **Perform a non-abstract task.** Some robotic tasks are sufficiently concrete that there is not much more to do than learn the world model. The prime example of this is the trajectory tracking task studied by Raibert, and similar tasks are in [Atkeson, 1989; Atkeson and Reinkensmeyer, 1989; Miller *et al.*, 1987].

- **Use a model-based pre-programmed controller.** This is a logical extension of the previous approach. Given an abstract problem, design a model-based controller to achieve the problem. Such model-based controllers can be simple and unsophisticated. A recent example of this is the controller for a visual tracker designed in [Miller, 1989], which is guided by a program that can be expressed as a short set of decisions and feedback rules. The top-level programs are of a sufficiently simple form that there may be some mechanism to generate them automatically.

- **Learn an evaluation function.** As well as learning the world model, an evaluation function on world states can be learned. An *evaluation function* is a mapping of the form

$$\text{EF} : \textbf{State} \rightarrow \Re \tag{3.7}$$

  Conventionally, the lower EF(s), the better the state s. The world model in conjunction with the evaluation function can be used to choose actions. Given a current state, the set of possible actions is consulted, and for each candidate action the evaluation is computed of the predicted resultant state. The action is chosen which minimizes the predicted evaluation. This is efficient provided the number of possible actions is not large.

This method is used by [Connell and Utgoff, 1987] to balance a pole, though in this case the world model is not learned, but estimated from the single previous state transition. A mapping is learned from states to the expected time to disaster. Learning evaluation functions has also been used in other domains such as puzzle and game learning, where the world model is trivially available and does not need to be learned [Samuel, 1967; Rendell, 1983].

3.3 The Important Issues for Learning Control

3.3.1 The Curse of Dimensionality

Realistic systems, whether they are learning action maps, or models of the world, should be able to cope with domain dimensions between approximately zero and eighteen (eighteen, because a direct drive six-jointed arm has twelve dimensions to its state space and six dimensions to its action space). However, much work has been restricted to learning task dimensions between zero and four. As we will see, many learning representations and convergence times become exponentially worse with increasing dimensionality. Approaches which denumerate all possible actions similarly blow up with increasing dimensionality of the action space.

These problems are compounded when, as is usually the case, the variables of state space and action space are continuous. For example, it is generally not known in advance at which level it is safe to quantize, or whether the quantization levels should vary.

The problem of dimensionality has rarely been directly addressed in the learning control literature. It is generally dealt with in one of the following ways.

- **Assume the control spaces are small and denumerable.** This is the assumption of stochastic automata [Fu, 1970], and of systems which make brutal quantizations to state spaces [Michie and Chambers, 1968; Barto *et al.*, 1983; Christiansen *et al.*, 1990], and commonly for reinforcement learning research [Kaelbling, 1990b; Sutton, 1990]. Similarly action spaces are often small (for example the classic pole balancer has only two actions). This is a reasonable approach for initial investigations of other aspects of learning control, but there is no doubt that it is useful, at some point, to take these initial approaches up to bigger problems.

- **Assume there is underlying, discoverable, structure in the problem.** To generalize in any way it is essential to have this assumption in some form. However, the strength of the assumption can vary very greatly. Parameterized mapping learners, which are described in Section 5.2, and which include polynomials and neural nets, use a strong form of the assumption [Minsky and Papert, 1969; Jordan and Jacobs, 1990]. Unless chosen with foreknowledge of the structure of the world model, there is no guarantee that any possible set of parameters could produce

a mapping which would adequately model the data. Decision tree classifiers [Quinlan, 1983] make a weaker assumption—that the domain can be split up into a fairly small number of large hyperrectangular regions in which the classification is constant. This feature is common with the approach of [Salzberg, 1988] and [Aha *et al.*, 1990], which both learn using the nearest neighbour generalization, but which use the assumption that classification regions can be characterized by a small number of well chosen example points (*exemplars*).

- **Only learn about one task.** Even if the state space is eight-dimensional, if only one trajectory is required, then the behaviour of the world need only be learned along one one-dimensional strand. This is the approach used by adaptive controllers for robot arms [Phan *et al.*, 1990; An *et al.*, 1988]. It was also used in [Miller *et al.*, 1987]. The dimensionality of the model is thus brought down to one.

- **Only learn small sub-areas of the task.** This is a natural extension of the previous approach, which learns small regions of the domain, but not as small as a one-dimensional strand. Even for an entirely repetitive task it is usually important to know about behaviour which is close to the solution of a task, but which is not actually in the solution of the task. This is in order to compensate for unpredictable deviations. Thus the controller's tactics in learning a task are to try to keep the experiences clustered around a fairly low dimensional, task-specific, subspace. In [Miller, 1989] this goal is stated. This idea is demonstrated in [Clocksin and Moore, 1989], in which a hand-eye coordination relation is learned for a five-jointed arm, but learning is biased to explore a two-dimensional subspace sufficient to reach all observed positions.

3.3.2 Variable Resolution

The ability to concentrate on particularly important areas of the control space requires a suitable choice of mapping representation. This aim is particularly important given the conclusion of the previous subsection—that the only defence against the curse of dimensionality without assuming extra domain knowledge is to concentrate on task-specific sub-areas.

This issue is considered in [Simons *et al.*, 1982] in which array boxes are recursively partitioned to increase resolution where necessary and also in the work of [Connell and Utgoff, 1987] which learned to balance a pole without needing to quantize the state variables.

3.3.3 Modularization

In almost all technological professions, large systems are broken down into smaller components—typically in a hierarchy. It is desirable to achieve this with learning control systems. The

result would be a group of simultaneous learning controllers, with some abstract controllers making use of other concrete controllers. There are two issues here:

- How should the hierarchy be organized?.
- How could the organization be achieved automatically?

The first question has been mentioned in several places [Fu, 1970; Sutton, 1990], but has not been discussed in detail. The exception to this is the architecture proposed in [Albus, 1981]. The second problem is interesting but very difficult, and has not been addressed for hierarchical structures with modules as complex as learning controllers.

3.3.4 Disorder in the Environment

The consequence of a disordered environment is that individual observations may not be reliable. The following are reasons that an environment may be disordered.

- **Noisy environment.** What the controller perceives is randomly perturbed from what actually happens. In this case the solution is to use local averaging of data, the implementation of which depends entirely on how the mapping is represented. Representations of mappings are discussed in Chapter 5.
- **Non-deterministic environment.** A simple approach to this problem would be to treat the non-determinism as noise. For interesting forms of non-determinism this is inadequate because the variation and probability distribution of the mapping being learned can also be valuable information for the controller. The tray tilting robot of [Christiansen *et al.*, 1990] learns to use actions which have minimal non-determinism in preference to unreliable actions. Non-determinism in the learning of action maps is also considered by [Kaelbling, 1990b; Sutton, 1990].
- **Non-stationary environment.** This problem has been investigated by [Raibert, 1978b; Miller *et al.*, 1987; Moore, 1990] for the case where the world which is being modelled is perturbed, requiring a quantitative change in the control strategy. A much harder problem is discussed in [Sutton, 1990], in which the control strategy can undergo a qualitative change. Sutton's DYNA-Q system is described in Section 3.4.

3.3.5 State Identification

Both action maps and world models need to be able to detect the state of the system. At this point it is worth recalling the definition of a system's *state*. Imagine that we can detect a certain amount of information about the current configuration of the system. If this information, combined with any proposed sequence of actions is *in principle* sufficient to determine future behaviour of the system, then the information is a representation of the system's state.

Most work in the field assumes the information provided to the system is sufficient to determine state, thus requiring a certain (although small) amount of world knowledge from the system designer. A partial solution to the case where the important aspects of state are unknown is proposed in [Simons *et al.*, 1982; Farmer and Sidorowich, 1988]. Another approach is suggested in [Vogel, 1989].

3.3.6 Experimenting

When the controller is learning, it needs to generate a diversity of experience. Methods to achieve this fall into three categories.

- **Use a teacher.** The role of a teacher is not simply to tell the system what should be done to perform the task, but can also be to guide the system to areas of the state space which are judged to be profitable to explore. The most common form of teacher has been a naive servo (linear feedback) controller [Atkeson and Reinkensmeyer, 1989; Miller, 1989; Miller *et al.*, 1987] which directs the experience towards areas of the state space which lie close to the solution. Extra domain knowledge of the structure of the world model is required to provide such a teacher.

- **Use randomness.** This is the most common approach to gaining experience. Recent examples of its use have been [Mel, 1989; Zrimec and Mowforth, 1990].

- **Estimate the utility of information-gain.** This has been recently investigated thoroughly by [Kaelbling, 1990a; Kaelbling, 1990b]. This work uses a statistical heuristic called Interval Estimation to choose actions which are likely to achieve reward, but which avoid getting stuck on repeated application of a known mundane action when there are superior actions with little experience available. The work investigates (i) algorithms in which there is immediate reinforcement, and (ii) delayed reinforcement (so that the choice of action is not only motivated by the next state, but perhaps by many states in the future). It also copes well with very non-deterministic environments. Choosing actions using heuristics which include the benefits of information gain has also been investigated by [Christiansen *et al.*, 1990; Sutton, 1990].

3.3.7 Inductive Learning

The discussion, and literature reviewed in this section, has been considering the design of a controller to perform well in its environment. There are other goals of learning, and one important one is to *explain* the environment. It can be argued that unless this is achieved, truly complex tasks will always require human intervention. Furthermore, it has been argued by [Michie, 1989; Sammut and Michie, 1989] that learning systems will not be accepted commercially unless their decisions can be understood by the human users.

3.4 Learning Robot Control: Recent Work

3.4.1 Connell and Utgoff: Variable Resolution Pole Balancer

In [Connell and Utgoff, 1987] an evaluation function was learned for the classic pole balancing problem (described in Section 3.2.1). The evaluation function was

$$\underbrace{\text{Cart-Pole state}}_{\textbf{State}} \rightarrow \textit{Desirability} \tag{3.8}$$

As mentioned in Section 3.2.3, an evaluation function in conjunction with a world model can provide the same functionality as an action map, but in this experiment no world model was used. Instead the evaluation of the next state if the most recent action were repeated is obtained. This is obtained using the behaviour of the most recent action in the previous state as a guide to how it would alter the current state. If the predicted evaluation is a decline then the alternative action is automatically used without predicting its consequences.

The evaluation function is represented by an explicit record of experienced states and interpolated using Shepard's method, described in Section 5.2. The evaluation function is not updated using the relationship between subsequent states, but according to an ad-hoc analysis of the 100 states prior to the collapse of the pole. However, with an appropriate choice of parameters it does quickly learn to balance the pole.

3.4.2 Miller: Learning World Models using CMAC

Recent work by W. T. Miller and colleagues [Miller *et al.*, 1987; Miller, 1989] has used CMAC [Albus, 1975a; Albus, 1975b] to model the world. The model is then used in conjunction with a teacher, in the form of a simple linear feedback controller, to improve performance. Two investigations have been reported in the literature.

- **Learning to track dynamic trajectories.** This work learned the inverse world model

$$\underbrace{\text{Joint angles} \times \text{Joint velocities}}_{\textbf{State}} \times \underbrace{\text{Joint Accelerations}}_{\textbf{Behaviour}} \rightarrow \underbrace{\text{Joint Torques}}_{\textbf{Action}} \tag{3.9}$$

  for a simulated two-jointed robot arm. Goal trajectories were defined in joint space coordinates. The main experiments used a repetitive trajectory, and were taught by a fixed-gain controller. The results showed quick improvement on the performance using the feedback controller alone. Experiments were also carried out with several trajectories to be learned and with changes in the environment. CMAC's behaviour was discussed, in particular the problems of too small an underlying memory.

- **Kinematic visual tracking of moving objects.** A three-jointed robot arm held a camera, pointing downwards at a conveyor belt. The work was motivated by the

advantages of being able to use world models in the same coordinate system as the task being learned. In this case the task coordinates were the visually sensed position and orientation of a plastic disposable razor and the sensed joint angles of the arm. The arm was controlled by requesting joint velocities which were obtained by independent servo motors in each joint. Both the forward and inverse world models were learned. The forward model was the relationship between the current joint angles, the current observed position of the razor, the requested joint velocities that were sent to the motors and the resulting change in the image position. The image position value was obtained by image processing, and consisted of three values: the x and y coordinates of the center of the razor's image and its orientation. The forward model was thus

$$\underbrace{\text{Joint angles} \times \text{Image position}}_{\textbf{State}} \times \underbrace{\text{Joint velocity}}_{\textbf{Action}} \rightarrow \underbrace{\text{Image change}}_{\textbf{Behaviour}} \tag{3.10}$$

and the inverse model was

$$\underbrace{\text{Joint angles} \times \text{Image position}}_{\textbf{State}} \times \underbrace{\text{Image change}}_{\textbf{Behaviour}} \rightarrow \underbrace{\text{Joint velocity}}_{\textbf{Action}} \tag{3.11}$$

The task was to keep the image of the razor fixed (which meant the arm had to move to keep the camera still relative to the moving razor). Experience was again provided by means of a teacher, a fairly complex position feedback controller. This required a substantial amount of domain knowledge, because the feedback was in joint space, whereas the tracking error was in image space. Two CMACs were used, one for each model. On each control cycle the forward model was used to predict where the razor would appear on the next cycle, and from this the desired image-position change was computed. The backward model was used to obtain a joint velocity to achieve the desired image-position change, and the feedback control signal was added. The results were again good, with a final average error of approximately a quarter of that obtained with the feedback control alone. Learning typically took about ten trials, with the razor being placed identically at the start of each trial. With random initial razor configuration, learning was considerably slower and less accurate.

3.4.3 Mel's MURPHY

This work [Mel, 1989; Mel, 1988] learned vision-based kinematic control of a real three-jointed planar arm. An interesting feature of the investigation was an "ecological" approach, in which the visual observations were kept in the raw sensed form of a 64 x 64 binary array. The control was again based on a learned world model:

$$\underbrace{\text{Joint angles}}_{\textbf{Action}} \rightarrow \underbrace{\text{Raw Image}}_{\textbf{Behaviour}} \tag{3.12}$$

The world model was forward. As Mel explains, this is generally the only valid direction in which to learn since the inverse model will usually not be a well-definable function. While true of most domains of other workers, this is particularly true of the kinematics of a *redundant* manipulator. A manipulator is redundant if there are multiple ways to move the gripper to a desired position (or desired position and orientation). The world model is learned by a period of random flailing of the arm. It is processed and represented by a *k*d-tree algorithm which has the behaviour of a neural-net (Chapter 6 introduces, describes and evaluates *k*d-trees).

After the world model is learned it is used to plan sequences of incremental joint modifications to reach target positions while avoiding visually observed obstacles. Mel stresses the importance of this planning taking place using the learned model rather than requiring real execution. The plan is a modified best first search using a visual distance heuristic. It is aided by a second learned world model—the inverse differential kinematics:

$$\underbrace{\text{Joint angles}}_{\textbf{State}} \times \underbrace{\text{Hand position change}}_{\textbf{Behaviour}} \rightarrow \underbrace{\text{Joint angle change}}_{\textbf{Action}} \tag{3.13}$$

This too is not a well-defined function, but Mel explains how to rectify this. The learning is successful, but computationally expensive (though much of the expense seems likely to be due to processing of the 64 × 64 images).

3.4.4 Atkeson's Memory-based Control

This work [Atkeson and Reinkensmeyer, 1989; Atkeson, 1989] learns to control a simulated dynamic robot arm to follow a trajectory and also learns corrections to a foot placement model for a simulated hopping robot. For the first experiment the model learned is the inverse world model

$$\underbrace{\text{Joint angles} \times \text{Joint velocities}}_{\textbf{State}} \times \underbrace{\text{Joint Accelerations}}_{\textbf{Behaviour}} \rightarrow \underbrace{\text{Joint Torques}}_{\textbf{Action}} \tag{3.14}$$

The representation is the explicit set of data points. A variety of generalizations are tried:

1. Nearest neighbour.
2. Local regression.
3. Local fitting to a quadratic surface.

The task is specified by a trajectory of joint angles and the experience is gained by means of a teacher—a linear PD-controller. The convergence is generally successful and quick, though the simple nearest neighbour generalization sometimes gets into "stuck states", in which performance is not improved. The likely reason is that an incompletely

learned inverse model can resuggest actions which are known to have failed. This problem is discussed in Section 5.1.

The rate of learning and final accuracy is seen to improve with increasing complexity of the method of generalization.

The foot placement task is learned as

$$\underbrace{\text{Hopper state}}_{\textbf{State}} \times \underbrace{\text{Velocity next step}}_{\textbf{Behaviour}} \rightarrow \underbrace{\text{Foot placement}}_{\textbf{Action}} \tag{3.15}$$

Instead of the learning the model directly, it is learned as an adjustment to a simple analytic world model. This difference mapping can be expected to be smoother than the direct model, and thus more easy to learn. Other aspects of the foot control are achieved using non-learning methods. The results show a marked improvement over using the simple world model, though stuck states are still a problem.

3.4.5 Zrimec and Mowforth's Block Pusher

In [Zrimec and Mowforth, 1990] a real robot learns the effects on a block of pushing it with the gripper of a robot. The block's position on a horizontal surface is observed visually, from above, before and after a smooth straight line robot movement. The world model learned is

$$\begin{array}{c} \underbrace{\text{Relative pos'n \& orientation of block}}_{\textbf{State}} \times \underbrace{\text{Relative movement of pusher}}_{\textbf{Action}} \\ \rightarrow \underbrace{\text{Change in relative pos'n \& orientation}}_{\textbf{Behaviour}} \end{array} \tag{3.16}$$

An interesting feature of this experiment is that there is no goal, simply an undirected aim to obtain knowledge. The representation of the mapping is a decision tree, which is able to produce a concise, human comprehensible, description of the mapping. Experimentation is by means of random movements.

In the same investigation the following further issues are discussed:

1. How to decide which variables are dependent on which others.
2. How to quantize the range of continuous variables.
3. Possible methods for automating 1 and 2.

3.4.6 Sutton's DYNA-Q Architecture

This work [Sutton, 1990] has only a rather small experiment, which leads to uncertainty as to whether the approach can scale up. However, there is discussion of a number of interesting and important issues for learning control. The DYNA framework assumes that the specification of tasks is only via reward, or reinforcement, signals. This assumption

makes the learning problem more difficult, because, for example it would forbid the use of feedback controllers as teachers. However, if DYNA learning is achieved then the system has a great deal of autonomy.

The system learns a world model, but also generates an evaluation function which estimates the relative qualities of states. The evaluation function is computed from the world model by dynamic programming [Burghes and Graham, 1980]. In practice, to avoid periods of wasted time in which the robot is planning without gaining extra knowledge for the world model, the dynamic programming occurs incrementally, in parallel with task execution.

The example is a simple maze domain with 56 discrete states and 4 discrete actions. The controller receives reinforcement of zero, except when it moves into a special goal state. If it reaches the goal state it is reset to its starting state. Thus, the tactics for maximum overall reward are to repeatedly trace the shortest path from start to goal, but even these abstract tactics are not given to DYNA in advance. Learning, and optimal behaviour, do indeed occur quickly after the first run.

Experimentation is random, but biased in favour of actions which are predicted to produce improved reward. As time progresses, the level of randomness decreases.

To cope with a changing environment, in which the current best set of actions might change, Sutton suggests adding an *exploration bonus* to the reinforcement which encourages rarely visited states to be occasionally visited in case there is possible improvement. This is demonstrated with a maze in which wall sections occasionally appear or disappear.

3.5 This Investigation

Here, I briefly mention which aspects of the field described in this chapter are investigated by this dissertation. I am concerned with making robot learning more practical, and so world model learning is used in preference to action map learning. A very important topic which is tackled is the curse of dimensionality, which has not been a specifically stated goal for other pieces of work. Part of the approach to this is to use a learning method with very variable resolution, and this has profound consequences for the representation of the mapping. The initial representation examined, while ideal for variable resolution, learning rate, and learning efficiency has poor noise tolerance and brittle performance in a non-stationary environment. The method would have a serious weakness could it not cope with these problems and so modifications are developed which do not lose its primary advantages.

A second critical aspect for avoiding dimensionality problems and for increasing the rate of learning is the nature of experimentation, and a method is developed which estimates the utility of information gain.

In order to compensate for the relative weakness of the autonomy of a world model learner (compared with an action map learner), the investigation also focuses on how

complex tasks can be modularized into a hierarchy of learning tasks. It also learns as abstract a model as it can by performing entirely in the *perceived* world.

3.5.1 Robustness

Another aim of this work is to have a robust learning method—one in which the system does not get into a situation which does not achieve the goal and in which further improvement does not occur. A learning system which is in such a situation is called *stuck*.

Some identified features which might cause sticking are:

- Insufficiently general class of learnable models. (Discussed for *parametric* methods in Section 5.2, and shown to be avoided in Section 5.3).
- Using only an inverse world model. (Discussed in Section 5.1).
- Failing to provide adequate experimentation. (Discussed in Chapter 8).
- Blame assignment errors in hierarchical learning systems. (Discussed in Chapter 9).

3.5.2 Issues not Addressed

Some important issues are not addressed by this investigation. In each case it is considered that the loss of the feature, while requiring more of the system designer, is also not generally critical for the autonomy of the system. These issues are state identification, learning a non-deterministic world model and providing an inductive ("simple") explanation of the world model. A further issue which is left to the system designer is scaling of state variables (discussed in Section 5.1).

Chapter 4

SAB Learning

This chapter introduces SAB Learning: the main idea which will be developed during the course of this dissertation. The chapter begins with robot problems which have no state, and introduces the concept of a Perception Function. It then extends this to dynamic state and the Perceived State Transition Function. Finally the sequence of actions taken by an SAB learner is described—the SAB Control Cycle.

4.1 AB Learning

Before considering full dynamic control of robots, we will examine the simpler problems of perception and geometry. An example of such a problem is hand-eye coordination.

The end-point of the arm is readily identifiable by some image processing. For example, one simple implementation is to subtract images of the arm obtained by moving only its gripper, and to then perform trivial statistics on the thresholded image. This obtains the *perceived coordinates* of the hand: the coordinates of the hand image.

The computer which receives these values can also *send* signals to the arm. These consist of a number of values, one sent to each joint. Each specifies, in *encoder units*, the angle (or, for a prismatic joint, the length) that the joint should take. When these signals arrive the joint angles are automatically and slowly adjusted by independent servomechanisms until the specified values are all achieved. Thus, for hand-eye coordination, the **Action** is the set of requested joint angles.

A typical hand-eye coordination task is to move the perceived hand position to a target perceived position. This can be specified directly by showing the goal to the controller. It is clear that to achieve the task the controller needs knowledge of the relationship between the perceived image coordinates and the raw action signal it sends. It is interesting to note that it *does not* necessarily need to know relationships between these and any absolute "real world" coordinate system.

This is an example of the general problem where a controller needs a relationship between the raw actions it is meant to supply and the perceived behaviour. In the cases

1. **Receive task specific goal behaviour b_{goal} from higher level of control.**

2. **Access the current world model to obtain a raw action a_{raw} which is predicted to be likely to achieve b_{goal}.**

3. **Apply action a_{raw}.**

4. **Observe actual behaviour b_{actual}.**

5. **Update the world model with the information that $a_{raw} \rightarrow b_{actual}$.**

Table 4.1: The AB Control Cycle

where there is domain knowledge that the relationship is a deterministic function from actions to behaviours let us call such a relationship the *Perception Function* (PF):

$$\text{PF} : \textbf{Action} \rightarrow \textbf{Behaviour} \tag{4.1}$$

The arm position task is an example of a PF because we have the knowledge that (i) given a set of joint angles the arm's real world endpoint position is determined and (ii) the cameras are fixed and so the perceived hand position on the camera is determined by the real world hand position.

To *learn* the PF would mean that the learning controller would have no knowledge about the relationship other than that it was a PF. A very general way of implementing such a learning controller is the *AB* Control Cycle, shown in Table 4.1.

Because we are dealing with raw action signals and raw perception signals (after crude image processing), the only substantial computation is in Steps 2 and 5. Requirements analysis and design of this computation is the part of main work of this dissertation. But before we consider this, in order to learn *dynamic* control, we must introduce the concept of perceived state.

4.2 The Perceived State Transition Function

A dynamic system has a *state*. Conventionally, when mathematical analysis is to be applied, the choice of the state representation is crucial to the tractability of the mathematics.

The conventional form of the differential equation derived from mathematical analysis is

$$\dot{\mathbf{x}} = g(\mathbf{x}, \mathbf{u}) \tag{4.2}$$

where $\mathbf{x}$ is the system's state, and $\mathbf{u}$ is the control input to the system [Burghes and Graham, 1980]. It is hoped that this equation will be simple, for example linear:

$$g(\mathbf{x}, \mathbf{u}) = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{4.3}$$

where $\mathbf{A}$ and $\mathbf{B}$ are constant matrices. The choice of state representation will strongly influence the simplicity, and hence tractability, of Equation 4.2. Here, let us call a state representation chosen to aid mathematical simplicity a *dynamic-model* state. The state is usually observed indirectly, by taking a series of measurements of the system, for example visual. Let us call the vector of these direct measurements the *perceived* state, $\mathbf{p}$, of the system (this is analogous to the definition of perceived behaviour in the previous section). The perceived state is transformed to and from the dynamic-model state by further mappings, the perception function PF and its inverse PF^{-1}.

$$\mathbf{x} = \text{PF}(\mathbf{p}) \text{ and } \mathbf{p} = \text{PF}^{-1}(\mathbf{x}) \tag{4.4}$$

The transformations PF and PF^{-1} are often at least as complex to model as the dynamics themselves. For example, in the case of a robot arm observed visually, PF is the composition of the inverse perspective transform and the inverse kinematics. The mathematical analysis derives the change in the perceived state $\mathbf{p}$ by transforming it to the dynamic-model state, then performing the dynamics model (Equation 4.2) and then finally transforming back to perceived space:

$$\dot{\mathbf{p}} = \mathbf{J}^{-1} g(\text{PF}(\mathbf{p}), \mathbf{u}) \text{ where } \mathbf{J}_{ij} = \frac{\partial \text{PF}_i}{\partial p_j} \tag{4.5}$$

$\mathbf{J}$ is the Jacobian matrix of the perception function PF.

In this investigation, we are not using mathematical analysis to perform the modelling, and so it is not necessary to perform these transformations explicitly, nor work out the inverse Jacobian matrix of the kinematics (called the inverse differential kinematics), nor indeed even necessary to invent a dynamic-model state representation. Instead, the perceived state dynamics are denoted by one mapping, the *perceived state transition function* (PSTF):

$$\dot{\mathbf{p}} = \text{PSTF}(\mathbf{p}, \mathbf{u}) \tag{4.6}$$

The PSTF is a deterministic function from the perceived *state* and raw *action* applied to the perceived *behaviour*.

$$\text{PSTF} : \textbf{State} \times \textbf{Action} \rightarrow \textbf{Behaviour} \tag{4.7}$$

One common example of perceived behaviour is the change in perceived state, but in Chapter 10 there are examples of other kinds of perceived behaviour.

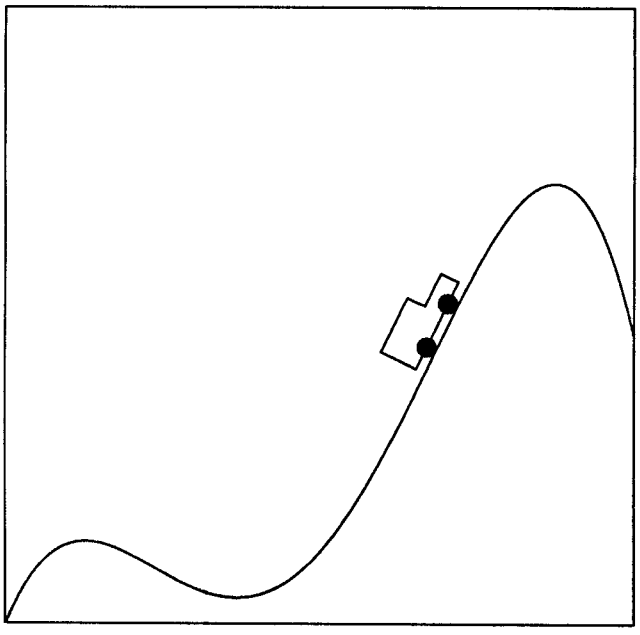

Figure 4.1

The "Mountain Car": a very simple dynamic system.

There are computational advantages to using the PSTF, but the greatest benefit, which will be described later, comes from the abstractness of perceived space, and the fact that it is in perceived space that the task descriptions themselves are likely to occur.

4.3 The Mountain Car Example

This is a very simple example which will be used for illustration during subsequent chapters. It is contrived so that the state space and control space are both one-dimensional. An automatic car drives along a one-dimensional track over a mountain range, as depicted in Figure 4.1.

The controlling station can sense the horizontal distance s that the car is from the start of the road. The controlling station can also send a signal u which specifies how far down the car's pedal should be pressed. Instantaneous changes to the pedal cause instantaneous changes to the car's speed. The speed depends on the pedal height, the gradient of the mountain where the car is, and also the altitude. Since these latter two features only depend on the car's location, we can deduce that the speed depends indirectly only on the pedal height and the distance along the road. The road is 1000 metres long and the pedal height varies between 0 and 10 centimetres. In the simulations that follow, this world will be used (none of this information is available to the controller):

- The height of the road (in metres) varies with distance in metres according to the

function

$$h(s) = 1000P(s/1000) \text{ where } P(x) = -8x(x^3 - 1.82x^2 + 0.95x - 0.16) \quad (4.8)$$

- The speed of the car along the road's surface is increased by decreasing the pedal height u. It is also increased by a downwards gradient or high altitude. The surface speed in metres per minute is

$$v_{\text{surface}}(s,u) = K_0\sqrt{1 - \frac{u}{10}} - K_1\frac{dh}{ds} + K_2 h \quad (4.9)$$

  where $K_0 = 140$, $K_1 = 10$ and $K_2 = 0.05$.

- The horizontal speed of the car depends on the surface speed and the current hill gradient:

$$v_{\text{horiz}}(s,u) = \frac{v_{\text{surface}}(s,u)}{\sqrt{1 + (\frac{dh}{ds})^2}} \quad (4.10)$$

  The horizontal speed varies between approximately -8m/min and 150m/min.

- The perceived state (distance travelled) is scaled uniformly in the range 0 to 10. The perceived action (pedal height) and perceived behaviour (horizontal speed) are similarly scaled. This arbitrary scaling exemplifies the fact that *SAB* learning need have no notion of the meaning of these variables. Writing s_p as the perceived state, a_p as the perceived action and b_p as the perceived behaviour:

$$b_p = \text{PSTF}(s_p, u_p) = 10\frac{v_{\text{horiz}}(10s/s_{\max}, 10u/a_{\max}) - b_{\min}}{b_{\max} - b_{\min}} \quad (4.11)$$

  where $b_{\max} = 150$m/min, $b_{\min} = -8$m/min, $s_{\max} = 1000$m and $a_{\max} = 10$cm.

The function $\text{PSTF}(s,u)$ can be graphed. The behaviour (speed) is a function of two variables: the state s (position) and the action u (pedal height). This is graphed in Figure 4.2. In this diagram the brightness denotes the behaviour—the brighter the tone, the closer the behaviour to the speed 5.7 units. This somewhat odd convention is adopted because we will later be studying a task in which the closer the speed to 5.7 units, the better. Thus, for example the graph shows that when $s = 6.5$ and $u = 3$ then the speed produced is 5.7. The bands below the central white band show increasing speed due to the pedal being closer to the floor. The bands above the white band show decreasing speed caused by the pedal being further from the floor.

4.4 The SAB Control Cycle

We are now ready to propose an enlargement of the learning controller of Section 4.1 which takes account of state. This is called the *SAB* control cycle, shown in Table 4.2.

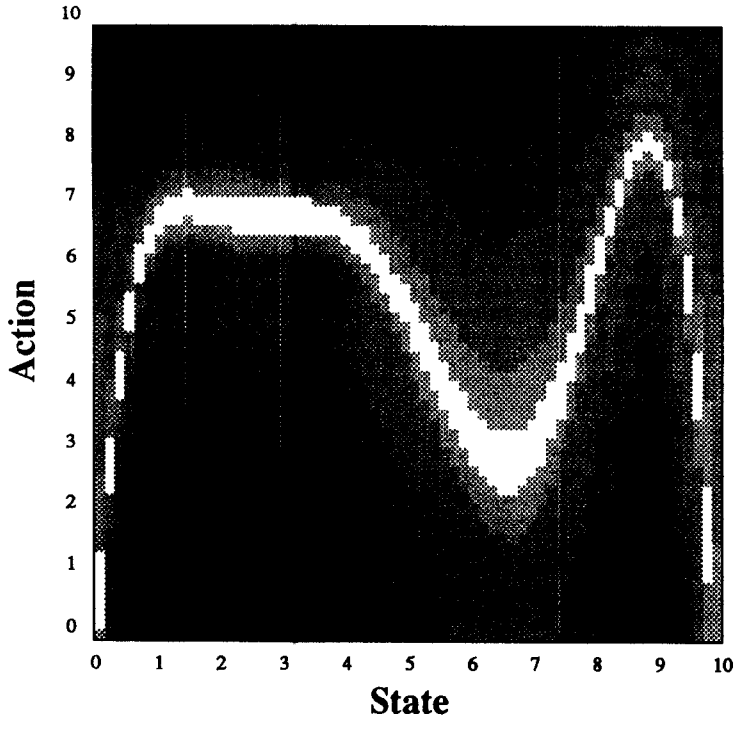

Figure 4.2

The Perceived State Transition
Function of the Mountain car. Brightness denotes the difference in speed from the goal speed of 5.7.

1. **Observe current perceived state $s_{current}$.**
2. **Receive task specific goal behaviour b_{goal} from higher level of control. The requested behaviour might depend partially on $s_{current}$.**
3. **Access the current world model to obtain a raw action a_{raw} which is predicted to be likely to achieve b_{goal} acting in the current state.**
4. **Apply action a_{raw}.**
5. **Observe actual behaviour b_{actual}.**
6. **Update the world model with the information that $(s_{current}, a_{raw}) \rightarrow b_{actual}$.**

Table 4.2: The *SAB* Control Cycle

It can now be seen that AB learning is a specific example of SAB learning in which the observed behaviour depends entirely on the chosen action, and there is no state to take into account.

Steps 3 and 6 are the crucial computations for the learning system. The performance of the robotic system will depend on the following:

- **Fast Learning.** It is desirable to minimize the length of time that the robot is training (that is, performing inadequately due to lack of knowledge). This is because of the expense of wasted robot time, and the possible hazard (or more reasonably the expense of precautions to avoid hazard) during this time of uncontrolled activity. Another reason for requiring a high rate of learning is that if changes occur in the environment, we wish the robot, through learning, to adapt to these changes reasonably quickly.

- **Cheap Learning.** Step 3 needs to take place within the timescale of the robot's dynamics, which is assumed to be a fraction of a second. As can be seen in Step 6 of the control cycle defined above, model updating has also been placed within the constraint in this investigation. There was an alternative to this decision: the update data could simply have been recorded during the execution of a dynamic task. Afterwards, when the robot was stable, the data would be used to update the world model. This alternative has two disadvantages: (i) It impacts the Fast Learning requirement during the initial trials, where we might have hoped to learn rough models within the very first moments of execution and (ii) it assumes that resting times will be available— in practice a large class of robotic tasks are under constant execution. An example is a balancing task.

- **Variable Resolution of Interest.** The state, action and behaviour spaces are multi-dimensional. This means that the number of significantly different states is enormous, and the number of significantly different state-action pairs is a magnitude greater still. A simple example would be a six-dimensional state space and a three-dimensional action space. Each dimension contains a continuum of values, but let us suppose, optimistically, that in each dimension only ten values are significantly different: then there are 10^9 different state-action combinations. At, say, six observations a second it would still take over five years to try each significantly different state-action pair even once. This illustrates an important restriction: we should not try to learn everything. Instead we should try to learn well and accurately in the regions of state-action space which are found useful for a task, and only learn a rough general picture for other areas.

The initial questions include: How should the world model be represented? How should generalization take place when asked to make predictions about things not experienced? Can the learning be all of fast, cheap and general? In the next chapter we examine a

simple powerful generalization and in subsequent chapters we will consider its efficiency and its ability to cope with disorder.

Chapter 5

Nearest Neighbour: Quick, Cheap Learning?

This chapter begins with a definition and description of the Nearest Neighbour generalization. Its history is also described. A survey of a wide variety of alternative learning techniques is then provided, along with a comparison to nearest neighbour. Later in the chapter the generality of nearest neighbour is formalized and proved, the speed of learning is analysed, and finally a variety of further aspects of nearest neighbour are discussed.

5.1 The Nearest Neighbour Generalization

The important components of a learning system are (i) an underlying representation of the concept being learned, called the *performance element*, (ii) an updating function which takes a new piece of data with which it updates the performance element and (iii) an accessing function which interprets the performance element to provide predictions, most importantly about data it has not seen.

The pieces of data which are presented are called, in this work, *exemplars*. For *SAB* learning they are triplets of data $(\mathbf{s}, \mathbf{a}, \mathbf{b})$ where $\mathbf{s} \in$ **State** is a perceived state, $\mathbf{a} \in$ **Action** is a raw action signal and $\mathbf{b} \in$ **Behaviour** is a perceived behaviour. Each is a vector of real numbers.

In the case of *SAB* learning there are actually two accessing functions which will prove useful. These are called *prediction* and *partial inversion*.

1. **Prediction:** Given a value $(\mathbf{s}, \mathbf{a}) \in$ **State** $\times$ **Action**, what is PSTF$(\mathbf{s}, \mathbf{a})$?
2. **Partial Inversion:** Given a value $\mathbf{s} \in$ **State** and a target value $\mathbf{b} \in$ **Behaviour**, what value of $\mathbf{a} \in$ **Action** (if any) will give PSTF$(\mathbf{s}, \mathbf{a}) = \mathbf{b}$?

The accessing function must *generalize*. This is what distinguishes a learning system

from a rote-memorizing system, which is not feasible in a high dimensional or continuous control space because there is not enough time to have every possible experience even once. The type of generalization depends on the learning application, and in particular can vary in strength. A very strong generalization, for example, might be to assume that the underlying function being learned is linear. Then the generalization could consist of linear regression from all the data received so far. A strong generalization can learn very quickly, but only provided that the underlying assumption is accurate: otherwise learning will not occur.

The nearest neighbour lies at the other extreme: it is a very weak generalization, based only on the assumption that the function is generally continuous (this is formalized later in this chapter). For the nearest neighbour the performance element is very simple: it is the explicit set of all the exemplars which have been received. In the following text this set is called **E**. After n observations it has value

$$\mathbf{E} = \{(\mathbf{s}_1, \mathbf{a}_1, \mathbf{b}_1), (\mathbf{s}_2, \mathbf{a}_2, \mathbf{b}_2), \ldots, (\mathbf{s}_n, \mathbf{a}_n, \mathbf{b}_n)\} \tag{5.1}$$

The update rule is also very simple: $\mathbf{E} := \mathbf{E} \cup \{(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})\}$.

The access rule uses the nearest neighbour generalization: To predict PSTF$(\mathbf{s}, \mathbf{a})$, find the $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i) \in \mathbf{E}$ for which $(\mathbf{s}_i, \mathbf{a}_i)$ is closest to $(\mathbf{s}, \mathbf{a})$. The predicted behaviour is $\mathbf{b}_i$. The notion of "closeness" is provided by the Euclidian distance metric, with the components of $\mathbf{s}_i$, $\mathbf{a}_i$ and $\mathbf{b}_i$ all scaled uniformly from their maximum ranges to the range $[0, 1]$. This is discussed shortly.

Let us consider an example. Figure 5.1 shows the interpretation after just four exemplars have arrived. In this example all spaces are one-dimensional and

$$\mathbf{E} = \{(1, 2, 5), (2, 2, 7), (4, 6, 7), (7, 4, 5)\}. \tag{5.2}$$

The figure graphs "behaviour predicted" against "perceived state" on the horizontal axis and "action" on the vertical axis. For example, all the points in the polygon surrounding the leftmost '5' correspond to state-action pairs which will be predicted, according to the nearest neighbour generalization, to result in behaviour **5**.

Given **s** and a desired **b**, partial inversion could be accomplished by finding a $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i) \in \mathbf{E}$ such that $\mathbf{s} = \mathbf{s}_i$ and $\mathbf{b} = \mathbf{b}_i$. However, in general such an exemplar will not be available. Instead, the exemplar with the nearest $(\mathbf{s}_i, \mathbf{b}_i)$ to $(\mathbf{s}, \mathbf{b})$ can be expected to have a good $\mathbf{a}_i$, *provided* that the predicted value of PSTF$(\mathbf{s}, \mathbf{a}_i)$ is $\mathbf{b}_i$. This proviso is important. Figure 5.2 provides an alternative view of the same exemplar set with "recommended action" plotted against "perceived state" on the horizontal axis and "required behaviour" on the vertical axis. First, let us consider an example where partial inversion is successful:

> If the current state is **0** and we require behaviour **5** then the nearest neighbour is the leftmost '2' exemplar. This recommends action **2**. Indeed, looking at the earlier prediction figure, this is a sensible recommendation because it predicts that applying action **2** in state **0** will give the desired behaviour **5**.

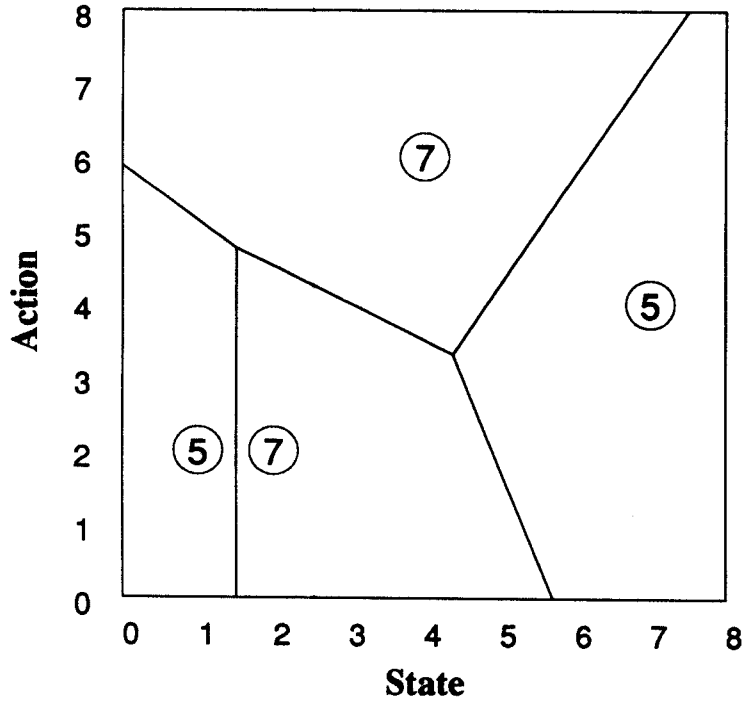

Figure 5.1

The nearest neighbour generalization predicting behaviour as a function of state and action.

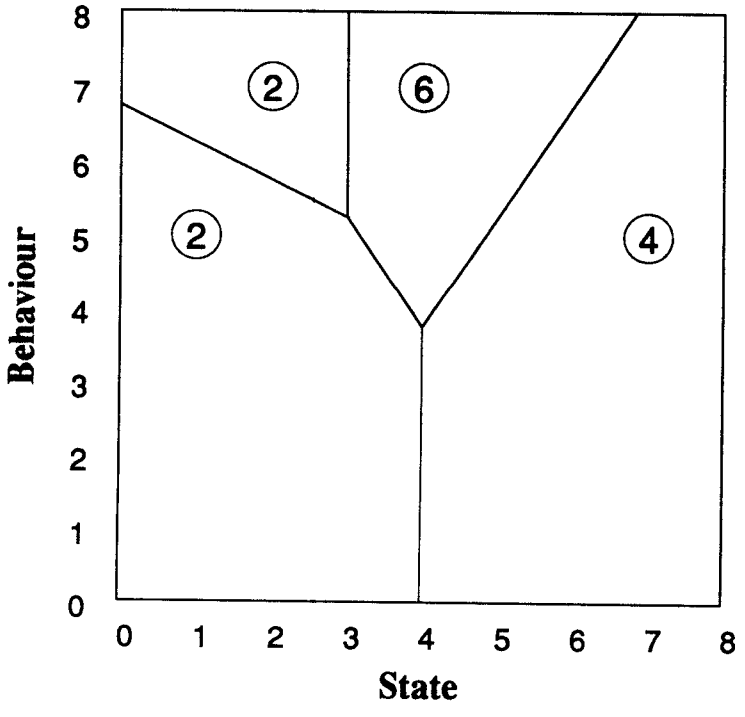

Figure 5.2

The nearest neighbour generalization performing partial inversion: the recommended action is derived as a function of state and desired behaviour.

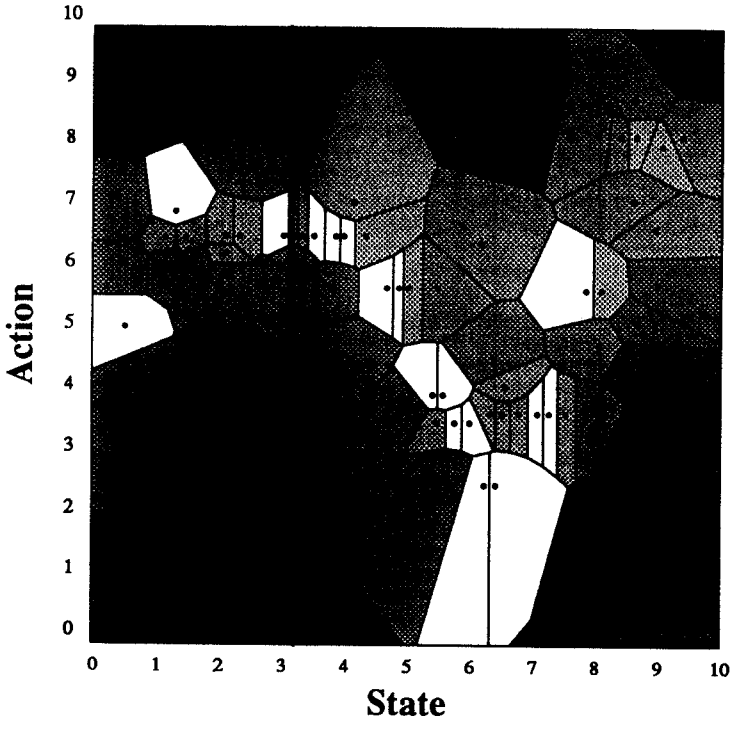

Figure 5.3

The nearest neighbour interpretation of the Mountain car PSTF after 100 experiences.

However partial inversion is not generally so successful:

> If we were in state **2** and required behaviour **5** then again, examining the lower diagram, we would be recommended action **2**. But if we examine the prediction we can see that we already know what happens if we apply action **2** in state **2**: we get behaviour **7**. Thus partial inversion used without prediction makes a serious mistake: and worse, it won't learn anything from the mistake, it will simply re-record the exemplar (2,2,7).

The conclusion is that partial inversion, which is closely related to associative memory lookup, is not necessarily valid for control choice using the nearest neighbour generalization (further, it seems likely that the same problem would occur for other generalizations). Chapter 8 will examine in depth how we should choose actions.

Figure 5.3 shows the approximation to the Mountain car's PSTF after approximately 100 experiences of the world. These are not random experiences—they were obtained using the *SAB* action chooser of Chapter 8. The agreement between the function values shown in this figure and the real PSTF depicted in Section 4.3 is generally good.

5.1.1 The Euclidian Metric

The Euclidian metric, L_2, defines the distance between two k-dimensional vectors $\mathbf{c}_1$ and $\mathbf{c}_2$ as

$$| \mathbf{c}_1 - \mathbf{c}_2 | = \sqrt{\sum_{i=1}^{k} ([\mathbf{c}_1]_i - [\mathbf{c}_2]_i)^2} \tag{5.3}$$

where $[\mathbf{c}]_i$ is the ith component of vector $\mathbf{c}$. The choice of a Euclidian metric is natural, because it allows a measure of closeness of states and actions which is independent of the direction of the axes. However, the choice of metric is not important providing the neighbourhoods are of a reasonable shape. Other possibilities include the L_∞ metric (maximum component difference) or L_1 metric (Manhattan distance).

The components of the vectors are scaled within their maximum ranges to provide equal weighting. In robotics, the ranges of both sensors and actuators are generally explicitly known to the system designer. If not they can be discovered. In the absence of further information the decision to weight equally is arguably the most sensible. Empirical study in Chapter 10 shows that performance is not sensitive to the weighting. If the "fair" ranges were not known it *would* be possible to discover them from the data by *cross-validation*. This technique removes a small proportion (e.g. 10 %) of the exemplars and then tests itself by predicting their values using nearest neighbour lookup on the remaining exemplars. This is repeated for a variety of weight-sets, and the weight-set which results in the highest prediction accuracy is selected for use.

5.1.2 History of the Nearest Neighbour

The use of nearest neighbour for pattern classification was noted by [Duda and Hart, 1973] and [Friedman *et al.*, 1977; Bentley, 1980]. That it can also be used for prediction of real-valued functions has been noted by many investigators, for example [Cleveland, 1979; Omohundro, 1987; Kibler *et al.*, 1988; Farmer and Sidorowich, 1988]. It has generally been passed over in favour of local regression (described in Section 5.2) which is asymptotically more accurate, but less well suited to applications requiring high speed predictions.

The nearest neighbour can be found by means of a *k*d-tree representation in $O(\log N)$ time by an algorithm first proposed in [Friedman *et al.*, 1977]. This is described and evaluated in detail later in this thesis.

The use of fast nearest neighbour search has been recommended by [Omohundro, 1987] in a survey of a great variety of methods for efficient implementation of learned mappings and other learning behaviours. The $O(\log N)$ performance predicted by both the above references is somewhat optimistic in high dimensionality where, as has been pointed out by [Maclaren, 1989], the computational cost may be dominated by a very large constant term, exponential in the dimension. It is proved in [Preparata and Shamos, 1985] that nearest neighbour searching can be no faster than $O(\log N)$.

Omohundro also suggests that associative nearest neighbour search can be obtained by learning the function

$$\mathbf{In} \times \mathbf{Out} \rightarrow \{\text{True}\} \tag{5.4}$$

instead of

$$\mathbf{In} \rightarrow \mathbf{Out} \tag{5.5}$$

In [Maclaren, 1989] it is shown how a *k*d-tree-based nearest neighbour search can still be implemented in a non-Euclidian distance metric.

There have been a number of recent investigations which have actually used the nearest neighbour. For learning text to phoneme classification it has been studied by [Stanfill and Waltz, 1986], and for classifying objects according to shape it has been used by [Gottschalk *et al.*, 1989]. For general classification it has been investigated by [Salzberg, 1988] and [Aha *et al.*, 1990]. For learning robot control, it has been used by [Atkeson and Reinkensmeyer, 1989; Atkeson, 1989] (described in Section 3.4) and also in the investigations described in this thesis (see also [Clocksin and Moore, 1989; Moore, 1990]).

Salzberg extends nearest neighbour by trying to reduce the number of exemplars stored. This is achieved by only storing exemplars which had been incorrectly predicted prior to observation. The advantages of reducing the number of exemplars are the decreased memory cost, search cost and also a more concise, and thus more "explanatory", mapping representation. The disadvantage is a reduction in classification accuracy. Salzberg also adds to the classification regions caused by the nearest neighbour generalization with hyper-rectangular classification regions. He also discusses an important feature of nearest neighbour learning—it is *one-shot*: a concept can be learnt with only one presentation of an item of data.

The work of [Aha *et al.*, 1990] also modifies the nearest neighbour generalization with a similar feature to reduce the number of points stored. It also shows that for pattern classification with noisy exemplars this method can lead to poor predictions, and so supplements the algorithm with a mechanism for detecting and removing noisy exemplars. The work also proves the important result that all mappings with finitely bounded classification regions are PAC-learnable under the nearest neighbour generalization. PAC-learnability is introduced in [Valiant, 1984]; see also Sections 5.4 and 5.3, which will make use of some of their results.

5.2 Alternative Generalizations

I will now examine some alternative methods of generalizing from observed data in order to learn a mapping $\mathbf{In} \rightarrow \mathbf{Out}$. The set **In** is called the *domain* of the mapping and **Out** is the *range*. This investigation is strongly biased by the requirements for a robot learning system, discussed fully in Section 4.4: the learning should be fast, cheap and with the ability to adapt to different resolutions of interest.

The various representations originate from three families. **Neural** representations are inspired by the behaviour of the nervous systems of biological organisms. **Statistical** representations have also been used. Finally, **Algorithmic** methods have evolved from mainstream computer science. These labels are arbitrary and several representations could be argued to lie in more than one field, but the distinctions are useful because they reflect different motivations.

Another, orthogonal, distinction is whether the mapping uses real valued, numeric-discrete or symbolic inputs and outputs. For learning control we will generally be most interested in the former, but some representations described below can only use discrete or symbolic variables. A learning system in which the output is symbolic is generally referred to as a *classifier.*

In this section the following mapping learning methods will be considered:

- **Array** *(Algorithmic)*
- **Global Polynomial Regression** *(Statistical)*
- **Local Regression** *(Statistical)*
- **Perceptron** *(Neural classifier)*
- **Multiple Layer Neural Network** *(Neural classifier)*
- **CMAC** *(Neural)*
- **BACON** *(Algorithmic)*
- **Decision Trees** *(Algorithmic classifier)*
- Q **Nearest Neighbours** *(Algorithmic)*
- **Shepard's Interpolation** *(Algorithmic)*
- **Radial Basis Functions** *(Algorithmic)*
- **Genetic Classifiers** *(Algorithmic classifier)*

At the end of the section there is a comparative summary.

5.2.1 The Multidimensional Array

Real valued components of the domain vectors are quantized to bounded ranges of integers. The cell (or box) corresponding to each region of the domain typically contains statistics compiled from all the experiences in that region. In [Michie and Chambers, 1968] each cell contains the expected life-time and number of times each action was tried. In [Raibert, 1978] it contains the average parameters of the local simplified (assumed constant

state) arm dynamics. In [Christiansen *et al.*, 1990] it contains the statistics of probability distributions of state transitions.

The representation is computationally cheap because update is a simple array update and access is a simple array access. This has the advantage of simplicity of implementation but has several problems. First is the enormous memory needed if the control space is to have more than five or so dimensions and the variables be quantized to more than ten or so levels. Secondly there is the problem of generalization: if there are a very large number of partitions then to learn the whole state space it is necessary to record at least one value from each of these partitions. Finally, we have no ability to concentrate on interesting areas of the control space, or ignore uninteresting areas.

5.2.2 Polynomial Regression

This is a *parametric* method for generalizing functions, which means that it starts with an assumption that the function which we are approximating is a member of a family of functions which can be characterised by a finite number of parameters. A simple example of such a family of functions are the set of linear functions $\Re \rightarrow \Re$. Each function in this family is of the form $y(x) = mx + c$ and so can be characterized by two parameters m and c. The generalization from the data is to find the member of the family which best fits the data. The definition of "best fit" is typically the fit which minimizes the squares of the errors at the exemplars. For simple families this definition of best fit can lead to closed form solutions to obtain the best fit parameters from the data.

One such simple family is the polynomials of degree N for small N. A polynomial of degree N from $\Re^{k_d} \rightarrow \Re^{k_r}$ has

$$P(N, k_d, k_r) = k_r \begin{pmatrix} k_d + N \\ k_d \end{pmatrix} \tag{5.6}$$

parameters, which grows very fast with N and k_d. Would this be a suitable generalizing method? If the underlying function were a polynomial in the family then learning would require $P(N, k_d, k_r)$ observations before the function were learned perfectly. If the observations were noisy then the learning would still be robust, provided the number of observations significantly exceeds the number of parameters in the polynomial. The problems, however, are severe: for a robotic system the assumption that the underlying function is a polynomial is certainly wrong and so we can only hope that the best fit is good enough. This means large errors are likely in some areas of state space, and repeated data observations will not improve this. Furthermore, there is no opportunity to vary resolutions of interest. Update and access involve large matrix multiplications and inversions.

5.2.3 Local Regression

This is a method which explicitly remembers and uses all the data points which are presented to it. Learning consists merely of recording a set of observed data points $\{(x_1, y_1), \ldots (x_n, y_n)\}$. The method is based on the idea that locally a smooth function behaves like a low degree polynomial. Thus, to find the value of a point which has not been experienced, the local function is approximated by using those data points which are close to the query point. These local points have a low degree polynomial fitted to them. This scheme has been suggested in a variety of ways by a number of researchers:

- **Cleveland and Delvin's locally weighted regression.** It is suggested by [Cleveland, 1979; Cleveland and Delvin, 1988] who are particularly concerned with *noisy* data points. Each explicitly stored data point is assumed to be of the form

$$y_i = g(x_i) + \epsilon_i \tag{5.7}$$

  where ϵ_i is the noise signal with mean zero and g is a smooth function. To evaluate the smoothed value at x_i the method uses the nearest N_0 neighbours, which are then fitted to a low degree (e.g. degree one) polynomial. There are extra mechanisms to bias the contribution of nearby local points more strongly than further ones, and to remove distortions caused by anomalous outlying data points.

- **Grosse's LOESS.** A more recent extension of this is [Grosse, 1989] which is concerned with making the calculation more computationally tractable, even in high dimensions. However, the calculation is optimized for a fixed set of points, and no cheap way is suggested for adding individual new pieces of data.

- **Omohundro's "efficient neural net behaviour".** This is a slightly simpler scheme, proposed in a major survey of computational techniques for classification and regression [Omohundro, 1987]. For a k-dimensional domain, the $k+1$ nearest neighbours of a query point are obtained, and a linear model based on these neighbours is then used. The $k+1$ nearest neighbours can be obtained fairly quickly, using a *k*d-tree, described in detail in Chapter 6.

- **Other suggestions and applications** Similar schemes are proposed by [Kibler *et al.*, 1988] and [Farmer and Sidorowich, 1988]. Such local interpolation schemes have been used by [Clocksin and Moore, 1989] in learning manipulator perception and kinematics and [Atkeson, 1989] in learning dynamic manipulator trajectories. The latter reference also provides a more thorough overview of further literature in the field of local regression.

Possible advantages of this method are extreme accuracy with relatively few exemplars. The problem is the expense of finding suitable local exemplars. It is clear that at least k_d+1 are needed, where k_d is the dimensionality of the function domain. Unless the system

is to be extremely vulnerable to noise, many more are required. It is also necessary to find local points which *surround* the query point, otherwise the process is extrapolation instead of interpolation, which can lead to large errors. The regression needs the exemplars to be local before accurate interpolation is obtained. This produces a slower *initial* rate of learning than simple nearest neighbour.

Local interpolation is sufficiently expensive that our implementation for kinematics experiments had to be performed off-line from the robot task execution, which prevents learning occurring *during* activity. The work of [Atkeson, 1989] also performed interpolation off-line. This work used a repetitive trajectory, meaning local data points were accumulated around a one dimensional path through state space, which allows the local interpolation to be particularly effective.

5.2.4 Perceptrons

This, and the following two mapping learners are popular connectionist methods which have been used in successful learning control experiments. There is a very great literature of alternative neural network architectures. It should be remembered that neural networks are also used for purposes other than learning mappings such as dimension reduction, associative memory and probability equalization.

Perceptrons were described extensively in the book [Minsky and Papert, 1969]. They provide a simple way of learning classification mappings from real valued vectors to a finite set of symbolic values. It is an iterative learning method, in which internal weights are adjusted to reduce classification errors for each new input. If the classification regions are separable by a hyperplane then the mapping can be learned. Perceptrons can also be used to learn linear real-valued functions. The linearity requirement renders perceptrons unsuitable for general robot learning.

5.2.5 Multiple-layer Networks

These were popularized by [Rumelhart and McClelland, 1984]. They perform the same function as perceptrons but can learn non-linear discriminations. Given a fixed number of inputs and outputs there are a wide range of possible network configurations from which to choose. A configuration consists of N nodes linked together by M connections. Each connection (and occasionally nodes also) have a weight. Each node has a value associated with it—its *activation energy*. When an input is presented to the net, a selected subset of the nodes, called *input nodes* have their activation energies set according to the input. Adjacent nodes are affected according to combinations of the activations of their neighbours. According to these local rules the activation energies of all nodes change until the activations of another distinguished set of nodes, the *output nodes*, have been evaluated. This is the output of the system. During learning, when the wrong output is produced from the given input the weights can be adjusted to reduce the error on future

similar presentations. One popular algorithm to do this is *back propagation* which can be shown to provide a local best fit of the weights to the data. "Best" is used here in the sense of minimizing the least squares prediction error [Angus, 1989].

Neural nets are another example of a parametric method (the parameters are the connection strengths) and so, like low degree polynomials, suffer from the danger that given the net configuration no solution could possibly represent the mapping. The update and access of neural nets is cheap—in each case it is no worse than proportional to the number of weights. As well as needing to hope the best fit is good enough, the disadvantages include slow learning times. Slow learning occurs because a piece of data may have to be presented many times to direct the weights to a set of values which accurately predicts the data. There are also reported cases of the weight adjusting method becoming stuck.

5.2.6 CMAC

The Cerebellar Model Articulation Controller [Albus, 1975a; Albus, 1975b; Albus, 1981] learns real-valued mappings. Its functionality is similar to a multi-dimensional array, but with the following advantages:

- Updates are distributed over a neighbourhood to provide a generalization.
- Array cells are hashed to reduce storage requirements.

CMAC relies on the assumption that behaviours within a neighbourhood will tend to be similar, that is that the function is smooth. This is a far less restrictive assumption than that used for the parametric methods.

The functional behaviour of CMAC is similar to that obtained by radial basis functions, described below. In Section 11.1 of this dissertation the similarity is explained and there is a detailed description of how CMAC's behaviour can be efficiently captured using an algorithmic method similar to that used in this thesis.

5.2.7 BACON

BACON is a method in which the space of simple algebraic formulae is searched in order to find the algebraic formulae which best fits the data [Langley *et al.*, 1983]. This uses a heuristic search, in which small sub-formulae which display some regularity in some contexts, are combined until larger regularities are discovered. This has been very successful in learning some elementary laws of Physics and Chemistry.

It is necessary that there exists a simple formula to model the data. For our purposes this seems unlikely. The behaviour of real robotic systems is complex, and much complexity is derived from components which are far from mathematically ideal. Even the linear analytic models with a great deal of assumed simplicity have a complex closed form, indicating that it would be difficult for a BACON-like search to find the correct formula. Another problem for our purposes is that it is non-incremental: there is no clear way

of modifying the current theory as extra pieces of information arrive. However, for any non-incremental scheme, an incremental version is possible: simply remember all the data explicitly and have a continually running background process which processes all data so far. When it finishes executing it replaces the current performance element with the new one it has discovered and immediately starts executing again on the old data and the new data which arrived during the previous execution.

5.2.8 Decision Trees

These are classifiers, which learn functions from multi-dimensional numeric or binary domains to symbolic classifications. A well known example is [Quinlan, 1983]. The input domain is recursively partitioned. At each level the partition is binary, and splits the domain into the two subspaces in as informative a manner as possible. For example, a function of two binary variables, in which the result was greatly correlated with the first, and not correlated with the second, would be split in the first variable. The partitioning algorithm is formalized using the theory of information content. The notion behind decision trees is that they tend to find regular patterns in data. The classification regions correspond to rectangular hyperregions of the domain space. In their simplest form decision trees are not incremental, but recent work by [Utgoff, 1989] rectifies this.

The CART (Classification and Regression Trees) system [Breiman *et al.*, 1984] performs a similar function to both Local Regression and Decision Trees.

5.2.9 Q Nearest Neighbours

An extension to using the value associated with the nearest neighbour is to process the values associated with some small number of the nearest neighbours. Its advantage is protection against noisy data. This has been suggested in a wide variety of places, for example [Duda and Hart, 1973]. The most common suggestions are

- Take a vote among the values of the Q nearest neighbours, and use the most common. This is only appropriate for symbolic valued mappings.
- To use the median value of the neighbours. This is particularly appropriate for discrete-numeric values. It protects against occasional wildly inaccurate exemplars.
- To use the mean value of the neighbours. This is appropriate for numeric domains and provides some degree of protection against low-amplitude noise.

In each case, the learning rate is slowed somewhat in comparison to the rate of single nearest neighbour, because predictions continue to be influenced by non-local exemplars for a long time. The search for the Q nearest neighbours also takes significantly longer than for the single nearest neighbour.

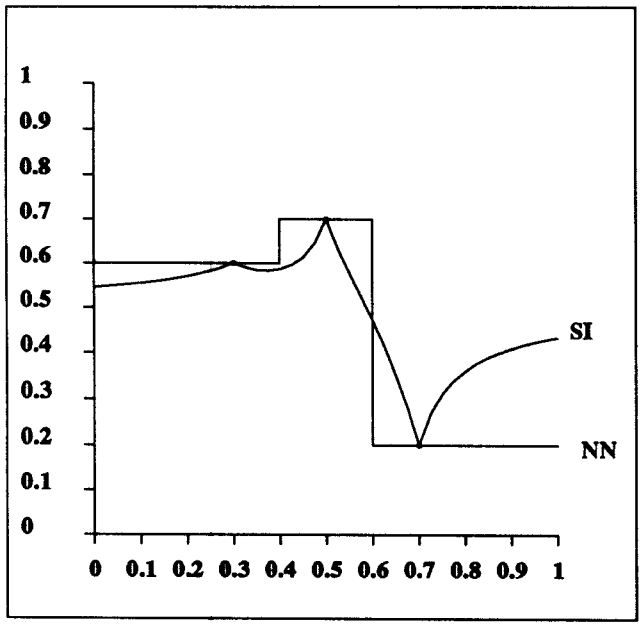

Figure 5.4

Interpreting a one-dimensional function from three data points (0.3 → 0.6), (0.5 → 0.7) and (0.7 → 0.2). This diagram compares the nearest neighbour (NN) generalization with a Shepard's Interpolation (SI) generalization.

5.2.10 Shepard's Interpolation

This method, described among others in a survey paper [Franke, 1982], records all data points explicitly, and when a query point is supplied, a weighted average of all the stored points is returned as the interpolated value. The contributions to the average are weighted so that stored points close to the query point make more of a contribution than those far from it. A typical weighting function is to have the weight of a stored point inversely proportional to the distance between the stored point and the query point. This method is depicted in Figure 5.4 in which three data points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ are stored, and Shepard's Interpolation (SI) for all other x coordinates gives

$$y_{\text{shep}}(x) = \frac{\sum_{i=1}^{3}(x - x_i)^{-1} y_i}{\sum_{i=1}^{3}(x - x_i)^{-1}} \tag{5.8}$$

For comparison, the interpolation produced by the nearest neighbour is also given.

Shepard's interpolation was used in [Connell and Utgoff, 1987] to learn an evaluation function for the pole balancing problem. The method has the problem of expensive access, because the contribution of every exemplar yet experienced needs to be included. Another disadvantage is that it does not generalize effectively at a large distance from any exemplars. Instead the interpolation tends to the average of all observed points.

5.2.11 Radial Basis Functions

A similar proposal to Shepard's interpolation is the use of radial basis functions, also described and evaluated in [Franke, 1982]. In this case contributions from all the exemplars are added together, instead of being averaged. Given a set of exemplars $\mathbf{E} = \{(x_1, y_1), \ldots (x_N, y_N)\}$ the prediction is defined as

$$y_{\text{rbf}}(x) = \sum_{i=1}^{N} A_i G(\mid x - x_i \mid) \tag{5.9}$$

The values $\{A_1, \ldots, A_N\}$ are weights which are computed from the exemplar set $\mathbf{E}$. A typical choice of the function G is the normal curve $G(x) = \exp(-x^2/c^2)$ for some constant c. The values of the A_i's are chosen to minimize some measure of error. This measure is typically the combination of the sum squared errors at the known exemplars and the "unsmoothness" of the resulting function (defined as the mean square second derivative of y_{rbf}) [Poggio and Girosi, 1989]. The computation of the A_i values makes update expensive, increasing linearly with the number of exemplars. In [Poggio and Girosi, 1989] a neural net based method is designed which has interpolative behaviour approximating that of Radial Basis Functions while only requiring a fixed amount of computation.

5.2.12 Genetic Classifiers

Genetic Algorithms [Holland *et al.*, 1987] are a recent optimization method in which spaces can be searched in a novel fashion which has been demonstrated to be particularly robust against capture by local minima. The search is most commonly in the space of fixed length strings of bits, but other syntactic domains can be used. The method keeps a fixed size population of partial solutions which are crossed over and mutated on the basis of their performance. This is analogous to some theories of biological evolution of genomes.

Among the many things which can be optimized are populations of *classifiers*. A classifier system is a set of *rules* which between them process an input vector to produce an output vector. The thing which is optimized is the accuracy of the set of rules. The optimization method consists of computing which rules should have credit for good classifications and which should be blamed for bad classifications. This computation is achieved by Holland's *bucket brigade* algorithm, described in the same paper. Individual rules tend to survive and be crossed-over if they are associated with successful predictions.

The power of genetic algorithm based optimization has been observed for a variety of domains (for example [Whitley, 1989; Gordon and Grefenstette, 1990]) but is not well understood. The vocabulary of the classifiers themselves is restrictive and more suited to discrete multi-input domains.

5.2.13 Summary

The mapping learning methods are shown in Table 5.1, which provides a rather terse description of the important features. The table is not meant to constitute an absolute judgement about the methods. Instead it provides a guide to how well each feature of each method was judged to suit the aims described earlier in this dissertation. An important detail is that the "Accuracy" field describes the expected accuracy, assuming that the function which we are attempting to learn is indeed learnable by the method under consideration.

	Accu-racy	Learning Rate	Gener-ality	Variable Resol'n	Noise Resist.	Update Cost	Access Cost	Memory Use
Array	Fair	Bad	Good	Bad	Fair	VG	VG	VB
Global Regr'n	Good	Good	Bad	Bad	VG	Poor	Poor	Good
Local Regr'n	Good	VG	VG	Good	Bad	Good	VB	Poor
Perceptron	Good	Fair	VB	Bad	VG	Good	Good	Good
Neural Net	Good	Bad	Poor	Poor	VG	Fair	Fair	Good
CMAC	Fair	Fair	Good	Poor	Fair	Poor	Poor	Fair
BACON	VG	Fair	Bad	Fair	Good	Bad	Good	Poor
Dec'n Tree	Good	Good	Fair	VG	Poor	Poor	Good	Poor
Q Nearest N'b'r	Fair	Fair	Good	Good	Fair	Good	Bad	Poor
Shepard Interp	Good	Good	Fair	Poor	Fair	Good	VB	Poor
Radial Basis	Good	Good	Fair	Fair	Fair	Good	VB	Poor
Genetic Class'r	Fair	Poor	Fair	Poor	Good	Poor	Fair	Fair
Nearest N'b'r	Fair	Good	VG	VG	Poor	Good	Poor	Poor

Table 5.1: Brief summary of a variety of generalizations discussed in the text.

Nearest neighbour is comparatively simple, and this is what permits real time access. Despite its simplicity it was selected as a robust learning method which best meets the goals for *SAB* learning. This decision is further justified in Sections 5.3—5.5 in which some remaining problems with nearest neighbour are also discussed.

5.3 The Class of Learnable Functions

Nearest neighbour is very general. It makes only a weak assumption about the mapping being learned. This can be described informally as the assumption that it is generally continuous. Here are examples of functions which will be seen to be nearest-neighbour learnable:

1. A continuous function over a closed interval.

2. A function with one or more asymptotes.

3. A piecewise uniformly continuous function with finite discontinuity.

In this section it is proved that nearest neighbour can achieve this, using the style of proof of [Aha *et al.*, 1990]. Let the mapping to be learned be $f : [0,1]^k \rightarrow \Re$. I shall define a broad class of function which includes all three given above. It will include functions which are generally continuous and generally smooth, but not entirely so. The rest of this section uses elementary, but somewhat detailed, mathematics and may be omitted if the reader is prepared to simply accept these claims of generality. It should also be noted that this discussion, for simplicity, will assume that exemplars and query points are drawn from a uniform probability distribution. In fact, the results will still hold for any other distribution.

5.3.1 Generally Continuous Functions

Given a function $f : [0,1]^k \rightarrow \Re$, and a value $\epsilon > 0$, define the *continuous domain* $D_\epsilon \subseteq [0,1]^k$ as the set of points in $[0,1]^k$ which are greater than or equal to distance ϵ from any discontinuity of the function f. Notice that from this definition the function f is continuous over all closed connected regions of D_ϵ. Define f to be *generally continuous* (GC) if

1. $$\text{As } \epsilon \rightarrow 0 \text{ then } \mid D_\epsilon \mid \rightarrow 1 \tag{5.10}$$

   where $\mid D_\epsilon \mid$ is the volume of the set D_ϵ (more formally, $\mid D_\epsilon \mid = \int_{D_\epsilon} 1 dx$).

2. The partition of $[0,1]^k$ induced by the discontinuities contains a finite number of disjoint regions.

Informally, a function is generally continuous if the portion of space "close" to a discontinuity diminishes to zero as the notion of "closeness" tends to zero. It can be shown that the three classes of function mentioned above are all generally continuous.

The set of GC functions is very large, and safely includes all analytic functions which have been used to model robot perspective, kinematics or dynamics, both quantitatively and qualitatively. It is not easy to imagine functions which are not GC, but one example is

$$f(x) = \begin{cases} 0 & \text{if } x \text{ is rational} \\ 1 & \text{if } x \text{ is irrational} \end{cases} \tag{5.11}$$

5.3.2 Nearest Neighbour can Learn Generally Continuous Functions

Any piecewise continuous mapping which satisfies the definition of GC can almost all, almost surely, be learned by nearest neighbour to any required accuracy. This statement can also be formalized in the manner of Valiant's work:

Imagine we perform nearest neighbour by drawing N random exemplars. Then for any probability $\alpha > 0$ and any proportion $\beta > 0$ and any accuracy $\theta > 0$ we can find an N such that:

With probability greater than $(1 - \alpha)$
For all points x in the domain except a proportion β
The nearest neighbour prediction of $f(x)$ differs from the value of $f(x)$ by less than θ.

This is proved using a result called the *coverage lemma* which we do not prove here. The coverage lemma is stated as follows [Aha *et al.*, 1990]:

The Coverage Lemma. For any $\epsilon > 0$, $\alpha > 0$ and $\gamma > 0$, if you draw

$$N(\epsilon, \gamma, \alpha) = \left(\left\lceil \sqrt{k}/\epsilon \right\rceil^k / \gamma \right) \times \log \left(\left\lceil \sqrt{k}/\epsilon \right\rceil^k / \alpha \right) \tag{5.12}$$

k-dimensional exemplars from any fixed probability distribution, then with probability greater than $(1 - \alpha)$ all but proportion less than γ of the domain will lie within distance ϵ of an exemplar.

A proof of the coverage lemma for $k = 2$ is given in [Aha *et al.*, 1990]. The learnability of nearest neighbour can now be proved. We need to achieve accuracy θ on all but a proportion β of the domain. To achieve this we will begin by fixing $\epsilon_1 > 0$ to create some continuous domain D_{ϵ_1} (later on we will find a suitable value of ϵ_1 for our purposes). D_{ϵ_1} is a set of one or more disjoint regions, none of which have discontinuities within the region or at the edge of the region. It is a standard result of mathematical analysis [Burkhill, 1978] that a function which is continuous within such a closed interval R is *uniformly continuous* within R:

$$\begin{aligned} &\forall \theta > 0 \quad \exists \tau > 0 : \\ &\qquad (\forall x, x' \in R \quad | \, x - x' \, | < \tau \;\Rightarrow\; | \, f(x) - f(x') \, | < \theta) \end{aligned} \tag{5.13}$$

So for each region R in D_{ϵ_1}, let τ_R be the value necessary to ensure that any points within distance τ_R of each other differ in function value by at most θ. We can choose a non zero value of ϵ which is less than τ_R for all regions (it is guaranteed non-zero because there are only a finite number of regions). We can also ensure that $\epsilon < \epsilon_1$. Thus if any two points of R are within distance ϵ then their values differ by less than θ.

For each region R let I_{ϵ_1} be an *internal connected region*: the set of points in R which are not within distance ϵ_1 of the edge of R.

$$I_{\epsilon_1} = \{x \in R : \forall x' \in [0,1]^k, \, | \, x - x' \, | \leq \epsilon_1 \Rightarrow x' \in R\} \tag{5.14}$$

Figure 5.5 provides an illustration. The domain has two discontinuities: one is along a horizontal line and the other is at a single point. There are two disjoint R regions (sets

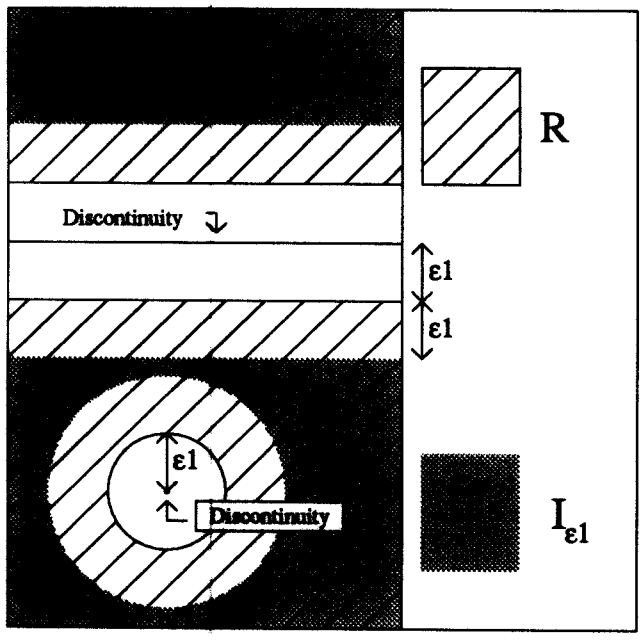

Figure 5.5

A two-dimensional domain with two discontinuities: one is along a horizontal line and the other is at a single point. The R regions are shown by cross hatching and the I_{ϵ_1} regions by gray shading.

of points further than ϵ_1 from a discontinuity) and thus two disjoint I_{ϵ_1} regions: each a subset of an R region.

From the definition of I_{ϵ_1}, if $x \in I_{\epsilon_1}$ then, for x' within distance ϵ of x, x' must be in the same region R (because it is closer than ϵ which has been defined to be less than ϵ_1), and so $\mid f(x) - f(x') \mid < \theta$.

We would like to take enough exemplars that we can ensure that all points in D_{ϵ_1} lie within distance ϵ of an exemplar, which would mean that for any point in any internal connected region I_{ϵ_1}, the nearest neighbour would have a value which differed by less than θ. Unfortunately, for any number of exemplars it is not guaranteed that all points lie within distance ϵ of an exemplar. However, by the coverage lemma we *can* ensure that with probability $1 - \alpha$, for all but proportion $\gamma = \frac{1}{2}\beta$, all points will lie within distance ϵ.

Now a suitable value of ϵ_1 can be chosen. It should be chosen to be small enough that all but proportion $\frac{1}{2}\beta$ of points lie within an internal connected region I_{ϵ_1} (the gray areas on the diagram). We are guaranteed that such an ϵ_1 exists because

$$\lim_{\epsilon_1 \to 0} \mid I_{\epsilon_1} \mid = \lim_{\epsilon_1 \to 0} \mid D_{2\epsilon_1} \mid = 1 \tag{5.15}$$

The necessary value of ϵ_1 is thus determined by β and the value of ϵ is determined by ϵ_1 and θ. This is adequate to complete the proof, because now, with probability $1 - \alpha$, all but proportion $\frac{1}{2}\beta$ of points within internal connected regions are predicted to accuracy better than θ by nearest neighbour. Furthermore, all but proportion $\frac{1}{2}\beta$ of all points in $[0,1]^k$ are in internal connected regions and so in total all but proportion β are sufficiently

accurate.

5.4 The Accuracy of Learning

The previous section proved that a very wide range of functions could be learned to arbitrary tolerance with nearest neighbour. But a question which is crucial for practical use is how many exemplars are required to achieve the tolerance? Conversely, how does the error decrease as the number of exemplars increases? The number of exemplars required to learn, with probability $(1-\alpha)$, all but proportion β of the domain to accuracy θ depends directly on the values of α, β and θ and indirectly on the function f itself. If, for a given function this dependency is exponential in any of $\frac{1}{\alpha}$, $\frac{1}{\beta}$ or $\frac{1}{\theta}$ then it can be argued that the function is not practically learnable [Valiant, 1984]. In Appendix B, in a proof slightly modified from those in [Kibler *et al.*, 1988] and [Omohundro, 1987], it is shown why, for uniformly continuous functions, the number required is polynomial in all three values.

Appendix B shows that the number of exemplars $n(\alpha, \beta, \theta)$ required is

$$n = \left(\frac{2}{\beta}\left\lceil\frac{G\sqrt{k}}{\theta}\right\rceil^k\right) \times \log\left(\frac{1}{\alpha}\left\lceil\frac{G\sqrt{k}}{\theta}\right\rceil^k\right). \tag{5.16}$$

Thus when learning uniformly continuous functions the number of k-dimensional exemplars required is better than $(k+1)$-nomial in $\frac{1}{\theta}$, linear in $\frac{1}{\beta}$ and sublinear in $\frac{1}{\alpha}$.

Appendix B also outlines why most other generally continuous functions should be expected to be learnable in polynomial time. However, even if the learning rate is polynomial, it can still be asked if after a practical amount of time the generalization will be to sufficient accuracy. The above analysis is worst case, and trying some numbers in Equation 5.16 indicates how pessimistic this worst case analysis is. Let us consider an example. The function f is three-dimensional and continuous with a maximum slope G of 1. We wish to learn the function over $[0,1]^3$. We are satisfied if, with probability 95%, over proportion 95% of the unit cube, nearest neighbour achieves an accuracy of $\frac{1}{3}$. Equation 5.16 tells us that the number of exemplars needed is no more than 72326. It should be immediately clear that this bound, although valid, is not close to the actual number one would expect to be needed. This feeling is supported by empirical evidence. The graphs in Figures 5.6 and 5.7 indicate that for one example of such a function,

$$f(x,y,z) = \frac{\sin(\pi x)\sin(\pi y)\sin(\pi z)}{\pi\sqrt{3}}, \tag{5.17}$$

the number of exemplars needed to achieve much better worst case accuracy over the whole domain is much less. These empirical tests were carried out for a variety of sizes of randomly generated exemplar sets. For each size, forty independent trials were run. Each trial estimated the average and worst case error by the average and worst case of 10,000 random nearest neighbour queries respectively. For a target accuracy of $\theta = 0.1$ the required number of exemplars is, in 95% of cases, less than 200 exemplars. Even this

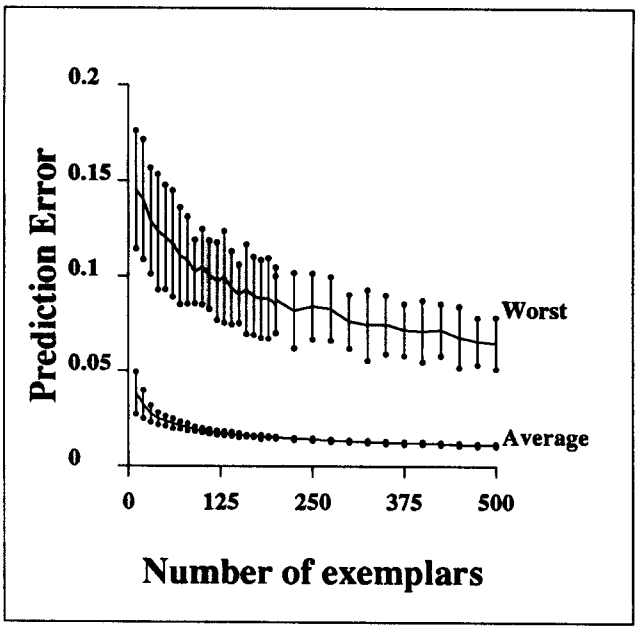

Figure 5.6

The worst case error and the average error plotted against the number of three-dimensional exemplars. These empirical trials were performed forty times; the vertical bars indicate the standard deviations of the data points (see Appendix A).

figure is grimmer than necessary, because we will be particularly interested in the *expected error*. In this case the empirical results indicate that only 10 exemplars are needed to bring the expected error to less than 0.1. Can formal analysis provide some measure of the expected error?

The answer is that it can, by estimating the expected distance to a nearest neighbour after n exemplars have been received. The following paragraphs review how this value is derived in terms of n and k, the dimensionality of the relation, under the assumption that the exemplars are distributed uniformly.

Imagine that we are looking for the nearest neighbour of a point at the centre of a k-dimensional sphere of radius 1 which contains n uniformly distributed random exemplars. We wish to evaluate $E^{n,k}$, the expected distance of the centre to the nearest neighbour. Write $Q^{n,k}(x)$ as the probability that all of the exemplars are further than distance x from the center of the sphere. The exemplars' locations are distributed independently, so

$$Q^{n,k}(x) = (\textbf{Prob}(\text{Any one given exemplar is further than } x))^n \qquad (5.18)$$

Because the distribution is uniform, the chance that one exemplar is further than x is the proportion of the sphere which is not in the smaller sphere radius x. This proportion is $1^k - x^k$. Thus $Q^{n,k} = (1 - x^k)^n$.

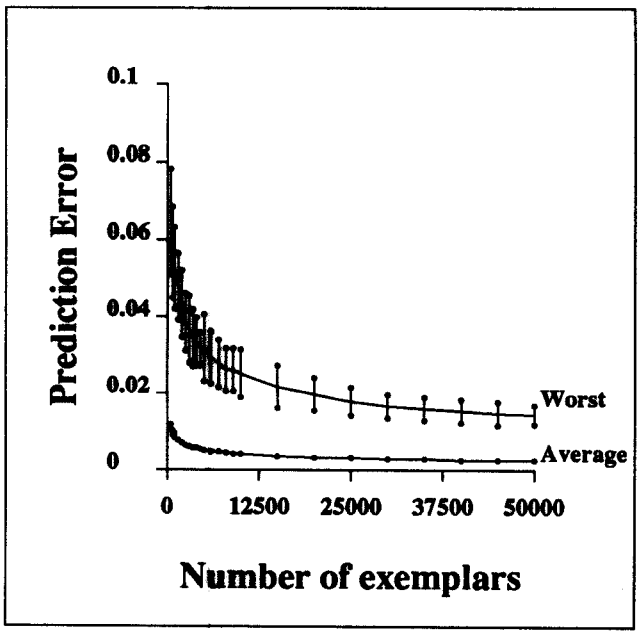

Figure 5.7

Plot of the same experiment as in Figure 5.6, but for larger exemplar sets.

Let $P^{n,k}$ be the probability distribution function

$$\begin{aligned} P^{n,k}(x) &= \textbf{Prob}(\text{Given } n \text{ } k\text{-dimensional exemplars, dist. of nearest} \leq x) \\ &= 1 - Q^{n,k}(x) \\ &= 1 - (1 - x^k)^n \end{aligned} \tag{5.19}$$

To derive the expected value of the nearest distance we need the probability density function

$$\rho^{n,k}(x) = \frac{dP^{n,k}}{dx} = nkx^{k-1}(1 - x^k)^{n-1} \tag{5.20}$$

The expected distance is

$$E^{n,k} = \int_0^1 x\rho^{n,k}(x)dx = nk \int_0^1 x^k(1 - x^k)^{n-1}dx \tag{5.21}$$

After performing this definite integration the result is

$$E^{n,k} = \frac{n!k^n}{(k+1)(2k+1)\cdots(nk+1)} \tag{5.22}$$

Let us assume that this estimate holds throughout the space $[0,1]^k$, rather than at the center of a sphere. This approximation becomes more accurate when there are more exemplars in the interval, because the probability of edge-effects is then reduced. Let us assume the function is uniformly continuous. The expected error in predicting the value of $f(x)$ depends on how far away the nearest exemplar is to x and on the magnitude of the slope G_x around point x. If we write $E^{n,k}_{\text{pred}}(x)$ for the expected prediction error at x,

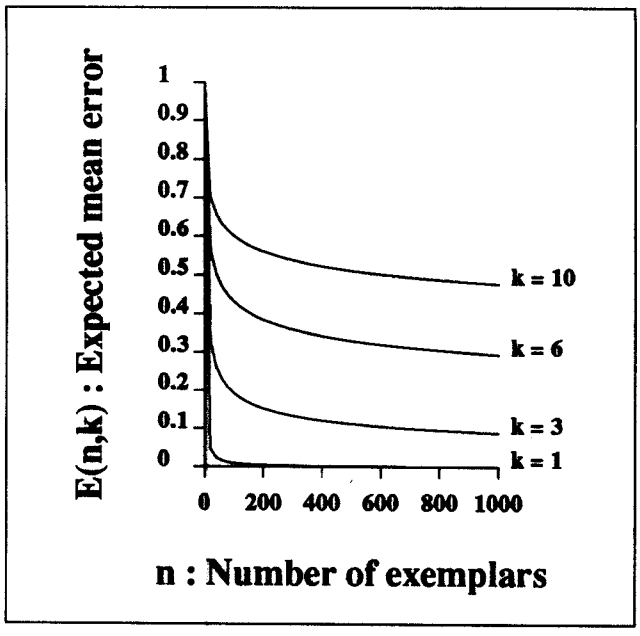

Figure 5.8

The expected accuracy of the nearest neighbour prediction for a uniformly continuous function with average slope 1. The dimensionality is k.

then we have $E^{n,k}_{\text{pred}}(x) = G_x E^{n,k}$. The expected error if we make a prediction of a random point chosen uniformly in the space $[0,1]^k$ is

$$A(n,k) = \int_{[0,1]^k} E^{n,k}_{\text{pred}}(x)dx = \int_{[0,1]^k} G_x E^{n,k} dx = E^{n,k} G_{avg} \tag{5.23}$$

Where G_{avg} is the average slope over the function. Assuming an average slope of 1, Figure 5.8 graphs the expected error based on this analysis.

It is clear that the error decreases, with a particularly fast initial improvement. It is also clear that large values of k have a devastating effect on performance. If we write $n(\frac{1}{\theta}, k)$ as the n required to achieve $A(n,k) = \theta$, then n is exponential in k, but for any fixed k can be shown to be polynomial in $\frac{1}{\theta}$.

$$n > \left(\frac{1}{\theta G_{avg}}\right)^{k+1} \Rightarrow A(n,k) < \theta \tag{5.24}$$

The low learning rate with large k would be a disaster if we were intending to try to learn everything. As noted earlier, we are assuming the data is received uniformly and accessed uniformly. If the function f were k-dimensional, but the data were all obtained from a subspace S of dimension k_{distrib} then it is clear we are really learning a function $f_S : [0,1]^{k_{\text{distrib}}} \rightarrow \Re$. The subspace need not be a linear projection for this to remain true. It is by using this that SAB-learning will attempt to dodge the curse of dimensionality, because in practice SAB-learning will *never* be trying to learn the entire space uniformly, simply because there is never enough time to do so. Instead, it will tend to remain close to

a low dimensional subspace of the control space. This low dimensional subspace might be 0,1,2 or 3 dimensional, but as we shall see later it need not generally be greater. Evidence for this, and examples of tasks where each of these underlying dimensionalities occur, will appear in Chapter 10.

With $k = 3$ or less, the accuracy is seen, both in the analysis and in the empirical experiment, to be adequate. The asymptotic convergence is slower than most of the other methods suggested in Section 5.2, but the other methods achieve greater speed by placing restrictions on the class of learnable functions which are far stronger than those for nearest-neighbour learnability. Furthermore, although they out-perform asymptotically, the merits of *initial* learning speed are still open to debate.

5.5 Nearest Neighbour: Discussion

Section 5.2 discussed some candidate generalizations which were considered as an alternative to nearest neighbour. This section describes in more detail why nearest neighbour was chosen, bearing in mind the performance criteria for *SAB* learning given in Section 4.4. Sections 5.3 and 5.4 discussed the generality and asymptotic learning speed of nearest neighbour. Here we will see that nearest neighbour indeed has further desirable features although it is not perfect. The acceptable imperfections will be identified. It will be the work of the two chapters following this one to overcome the other imperfections.

5.5.1 Initial Learning Speed

We shall see here that nearest neighbour is extremely quick at obtaining an initial rough model of important areas of the domain. There are three reasons:

1. For initial *SAB*-learning all that is needed is rough, qualitative control—very informally enough knowledge that the system's behaviour can be sent in the right direction instead of in the wrong direction. This means that the accuracy we need to attain rough control is perhaps a half or third of the maximum deviation over the function.

2. For a robot learning system the initial learning will all, necessarily, take place in a small region of the state space, although a large region of the action space might be explored. This is because there will not be time, in the first few moments of learning to travel very far from the start state. So, if we can consider the state effectively constant at the start of learning, then instead of learning on a

$$[Dim(\mathbf{State}) + Dim(\mathbf{Action})] \tag{5.25}$$

-dimensional domain we are learning on a

$$[Dim(\mathbf{Action})] \tag{5.26}$$

-dimensional domain.

3. The rate of learning is relatively much higher at the start of learning than at any other point, especially for high dimensionality.

An example is the control of the two-jointed arm considered in Section 10.2. The control space is six-dimensional, but as learning begins, the state is roughly constant. This leaves a two-dimensional action space over which the exemplars can vary, so the learning rate for $k = 2$ applies. Imagine the expected slope is 1, and we require an initial accuracy of 30%. Considering Equation 5.23, the number of exemplars needed is n where

$$0.3 > \frac{n!2^n}{(2+1)(2\times 2+1)\cdots(2n+1)} \tag{5.27}$$

which is satisfied by $n = 8$. The results chapter will detail how good the initial learning was for an empirical investigation of this system.

The initial learning is very fast because the learning is *one-shot*. This is in contrast to a connectionist or regression method in which several (and in some cases very, very many) pieces of data are needed before any meaningful predictions can be made.

5.5.2 Variable Resolution

The nearest neighbour generalization can vary its resolutions of interest very well. This means that areas of the control space which are especially important to a task can be learned to high accuracy, and uninteresting areas to low accuracy. It has already been described in several different contexts why this is so desirable:

- (Mentioned in Section 3.3) A robot will wish to concentrate on particular tasks, instead of learning generally.
- (Mentioned in Section 5.2) There is not enough time to visit each general area of the control space even once.
- (Mentioned in the previous section) The number of exemplars needed to reach a fixed accuracy is exponential in the underlying dimensionality, k_{distrib}, of the data points.
- (Will be discussed in Section 6.5) The computational cost of nearest neighbour is too high if we search in a uniform distribution of data.

The importance of varying resolutions was noted by [Simons *et al.*, 1982]. This work used a multi-dimensional array as the performance element, but to adapt resolutions it recursively broke selected boxes of the array into higher resolution sub-arrays.

Of all the generalizing methods discussed in Section 5.2, nearest neighbour is the best at varying its resolution of interest, and this comes for free. No extra implementation effort is required.

Local regression and radial basis functions also vary their resolutions, but because much more local data than just one neighbour is needed, the "rate of increasing locality" is very much reduced.

5.5.3 Simplicity

One advantage of a learning method being conceptually simple is that we can have confidence in its being robust. Its behaviour can be understood by its users. Nearest neighbour is a particularly simple method. It is intuitively clear how it works, and it has been demonstrated here to be sufficiently simple to be amenable to analysis in only a dozen or so pages. Here I will detail both advantages of simplicity.

- **Robustness.** A learning system initially can be expected to make errors, but if it continues to make the same error then, informally, one could claim that it is not learning. This can happen to some parametric methods. A common reason would be that the class of functions being parameterized did not include the actual function being learned. For some iterative methods, such as *back-propagation* [Rumelhart and McClelland, 1984], a second reason is that the iteration can get stuck in a local minimum. For nearest neighbour learning of any generally continuous function, this cannot happen because the representation is derived directly, instead of indirectly through iteration. Depending on the implementation of local regression, this can also make some very large prediction errors in regions of large curvature, especially if the local distribution of points is not dense or if it accidently extrapolates.

- **Understandability.** There are many occasions when one needs to know *why* a learning method made a particular prediction, particularly if there has been a significant error. Nearest neighbour can provide a fairly direct, understandable answer:

  > *Sorry. You see, in this very similar previous situation that decision seemed to work, so I thought it probably would here too. I'll know better next time.*

  Other methods based on statistical or connectionist models can only give a far less direct explanation:

  > *Sorry. You see, this is the algorithm and this is the data received so far, and so the execution resulted in this decision.*

  This comparison is somewhat unfair as some work is now being done to allow connectionist networks to explain their behaviour [Diederich, 1990].

In [Michie, 1989] it is argued that until learning systems can explain their decisions clearly they are unlikely to be accepted into commercial use. It should also be noted that the nearest neighbour explanation is inferior to that from a decision tree or from BACON, which can both give particularly succinct accounts of their current theory of the world.

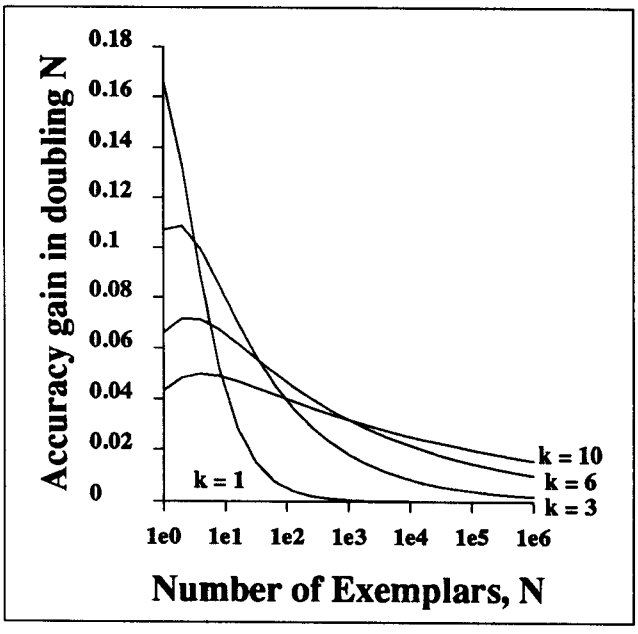

Figure 5.9

The horizontal (logarithmic) scale lists varying values of n, the number of exemplars. We plot the increase in accuracy, $\Delta\theta$, which would be obtained from doubling the number of exemplars.

$$\Delta\theta = E^{n,k} - E^{2n,k}$$

5.5.4 Computational Resource Costs

There is a significant fear that demands on the computer will increase linearly with n, the number of exemplars. The resources which might be badly affected are (i) time to update, (ii) time to access and (iii) memory requirements. Before discussing these cases it is worth considering just how large n will become. It is constrained by the length of time in which the robot learns. For example an extreme learning task might last three days with five SAB-cycles per second, in which case n would be 1,296,000 exemplars. But it is questionable whether would there be much point in continuing learning when n is in the millions. Each exemplar has, on average, only a one in a million chance of being accessed, and so there will not be much payoff from continued learning. This is borne out by Figure 5.9, which graphs the expected increase in accuracy obtained by doubling the current number of exemplars. It shows, for example, that if we have 100,000 three-dimensional exemplars from a function with average gradient 1, then taking a further 100,000 will decrease the expected error by approximately 0.01. For larger dimensionality there is a higher payoff in increasing the number of exemplars, but given the enormous time to double the number of exemplars, it will rarely seem worth it. However, instead of discontinuing learning, it is preferable to add new exemplars and delete old ones. This is also important to cope with a changing environment, and the topic is fully discussed in Section 7.3.

Thus, for these two reasons, we will not hold more than the order of a quarter of

a million exemplars in store. We can now make an estimate of the amount of memory which will be required. Assuming two bytes per field in the state, action and behaviour vectors, and assuming a twelve-dimensional state space, six-dimensional control space and six-dimensional behaviour space, with an overhead of another 30 bytes per exemplar (Chapter 7 will explain overhead requirements), the memory requirement to store $\frac{1}{4}$ million exemplars would be $19\frac{1}{2}$ megabytes. This is an amount of memory well within the virtual memory of modest computers in 1990, and is already within the on-board memory of powerful machines. It must, however, be mentioned that most parametric methods require a magnitude less memory. Multi-dimensional arrays require a magnitude more.

There is also the question of the time for update and access. Chapter 6 describes an implementation to make this sufficiently small. For low degree parametric methods, update and access are also cheap, but nearest neighbour is faster than the other suggested methods.

5.5.5 Disorder

I have not as yet mentioned a major advantage of parametric methods: they do not learn noise. Imagine we receive exemplars of the form

$$(\mathbf{c}_i + \boldsymbol{\delta c}_i \rightarrow f(\mathbf{c}_i) + \boldsymbol{\delta b}_i) \tag{5.28}$$

where $\boldsymbol{\delta c}_i$ and $\boldsymbol{\delta b}_i$ are noise signals distributed independently of i with mean zero. Parametric methods will, if they converge, converge to generalizations which give the correct, noise-free prediction. Nearest neighbour will do very badly: it will learn the noisy values exactly as they occur. Radial basis functions and a suitable implementation of local regression will cope, but they will need to include a large number of exemplars. In particular, local regression with only a few noisy exemplars is likely to have severe errors.

Chapter 7 explains how nearest neighbour is modified to average out noise, in a similar manner to Shepard's regression, but optimized for access efficiency. In doing so, it will be necessary to put an upper bound on the accuracy to which the function can be learned around discontinuities.

This investigation assumes that there is no information in the noise, or in the distribution of the noise: we simply want to get rid of it. This assumption is a consequence of the earlier assumption that the Perceived State Transition Function is deterministic. Some work has been done by [Christiansen *et al.*, 1990] on a robotic tray tilting system in which the observed world is non-deterministic and in which the distributions of observed behaviours need to be learned.

5.5.6 Knowing what one Knows

The nearest neighbour estimate can provide a measure of how accurate a given prediction is expected to be. This estimate is a function of the actual distance to the nearest neighbour. As we will discover later, this knowledge is important for deciding intelligently when

experimentation is desirable. Knowledge of knowledge can also be obtained directly from an array representation, and somewhat less directly from the other non-parametric methods reviewed earlier. It cannot be obtained from a parametric representation, although it is possible that the system could attempt to learn knowledge of knowledge explicitly.

Chapter 6

Kd-trees for Cheap Learning

This chapter gives a specification of the nearest neighbour algorithm. It also gives both an informal and formal introduction to the kd-tree data structure. Then there is an explicit, detailed account of how the nearest neighbour search algorithm is implemented efficiently, which is followed by an empirical investigation into the algorithm's performance. Finally, there is discussion of some other algorithms related to the nearest neighbour search.

6.1 Nearest Neighbour Specification

Given two multi-dimensional spaces **Domain** = $\Re^{k_d}$ and **Range** = $\Re^{k_r}$, let an *exemplar* be a member of **Domain** × **Range** and let an *exemplar-set* be a finite set of exemplars. Given an exemplar-set, **E**, and a target domain vector, **d**, then a *nearest neighbour* of **d** is any. exemplar $(\mathbf{d}', \mathbf{r}') \in \mathbf{E}$ such that *None-nearer*$(\mathbf{E}, \mathbf{d}, \mathbf{d}')$. Notice that there might be more than one suitable exemplar. This ambiguity captures the requirement that any nearest neighbour is adequate. *None-nearer* is defined thus:

$$\textit{None-nearer}(\mathbf{E}, \mathbf{d}, \mathbf{d}') \quad \Leftrightarrow \quad \forall(\mathbf{d}'', \mathbf{r}'') \in \mathbf{E} \quad |\, \mathbf{d} - \mathbf{d}' \,| \leq |\, \mathbf{d} - \mathbf{d}'' \,| \tag{6.1}$$

In Equation 6.1 the distance metric is Euclidean, though any other L_p-norm could have been used.

$$|\, \mathbf{d} - \mathbf{d}' \,| = \sqrt{\sum_{i=1}^{i=k_d} (d_i - d'_i)^2} \tag{6.2}$$

where d_i is the ith component of vector **d**.

In the following sections I describe some algorithms to realize this abstract specification with the additional informal requirement that the computation time should be relatively short.

Algorithm:	Nearest Neighbour by Scanning.
Data Structures:	
domain-vector	A vector of k_d floating point numbers.
range-vector	A vector of k_r floating point numbers.
exemplar	A pair: (**domain-vector, range-vector**)
Input:	**exlist**, of type **list of exemplar**
	dom, of type **domain-vector**
Output:	**nearest**, of type **exemplar**
Preconditions:	**exlist** is not empty
Postconditions:	if **nearest** represents the exemplar $(\mathbf{d}', \mathbf{r}')$,
	and **exlist** represents the exemplar set $\mathbf{E}$,
	and **dom** represents the vector $\mathbf{d}$,
	then $(\mathbf{d}', \mathbf{r}') \in \mathbf{E}$ and *None-nearer*$(\mathbf{E}, \mathbf{d}, \mathbf{d}')$.
Code:	
1.	**nearest-dist** := infinity
2.	**nearest** := undefined
3.	for **ex** := each exemplar in **exlist**
3.1	**dist** := distance between **dom** and the domain of **ex**
3.2	if **dist** < **nearest-dist** then
3.2.1	**nearest-dist** := **dist**
3.2.2	**nearest** := **ex**

Table 6.1: Finding Nearest Neighbour by scanning a list.

6.2 Naive Nearest Neighbour

This operation could be achieved by representing the exemplar-set as a list of exemplars. In Table 6.1, I give the trivial nearest neighbour algorithm which scans the entire list. This algorithm has time complexity $O(N)$ where N is the size of $\mathbf{E}$. By structuring the exemplar-set more intelligently, it is possible to avoid making a distance computation for every member.

6.3 Introduction to kd-trees

A *kd-tree* is a data structure for storing a finite set of points from a k-dimensional space. It was examined in detail by J. Bentley [Bentley, 1980; Friedman *et al.*, 1977]. Recently, S. Omohundro has recommended it in a survey of possible techniques to increase the speed of neural network learning [Omohundro, 1987].

A *k*d-tree is a binary tree. The contents of each node are depicted in Table 6.2. Here I provide an informal description of the structure and meaning of the tree, and in the following subsection I give a formal definition of the invariants and semantics.

Field Name:	Field Type	Description
dom-elt	**domain-vector**	A point from k_d-d space
range-elt	**range-vector**	A point from k_r-d space
split	**integer**	The splitting dimension
left	**kd-tree**	A *k*d-tree representing those points to the left of the splitting plane
right	**kd-tree**	A *k*d-tree representing those points to the right of the splitting plane

Table 6.2: The fields of a *k*d-tree node

The exemplar-set **E** is represented by the set of nodes in the *k*d-tree, each node representing one exemplar. The **dom-elt** field represents the domain-vector of the exemplar and the **range-elt** field represents the range-vector. The **dom-elt** component is the index for the node. It splits the space into two subspaces according to the splitting hyperplane of the node. All the points in the "left" subspace are represented by the **left** subtree, and the points in the "right" subspace by the **right** subtree. The splitting hyperplane is a plane which passes through **dom-elt** and which is perpendicular to the direction specified by the **split** field. Let i be the value of the **split** field. Then a point is to the left of **dom-elt** if and only if its ith component is less than the ith component of **dom-elt**. The complimentary definition holds for the right field. If a node has no children, then the splitting hyperplane is not required.

Figure 6.1 demonstrates a *k*d-tree representation of the four **dom-elt** points $(2,5)$, $(3,8)$, $(6,3)$ and $(8,9)$. The root node, with **dom-elt** $(2,5)$ splits the plane in the y-axis into two subspaces. The point $(3,8)$ lies in the lower subspace, that is $\{(x,y) \mid y < 5\}$, and so is in the left subtree. Figure 6.2 shows how the nodes partition the plane.

6.3.1 Formal Specification of a kd-tree

The reader who is content with the informal description above can omit this section. I define a mapping

$$\textit{exset-rep} : k\text{d-tree} \rightarrow \text{exemplar-set} \tag{6.3}$$

which maps the tree to the exemplar-set it represents:

$$\begin{aligned} &\textit{exset-rep}(\text{empty}) &&= \phi \\ &\textit{exset-rep}(< \mathbf{d}, \mathbf{r}, -, \text{empty}, \text{empty} >) &&= \{(\mathbf{d}, \mathbf{r})\} \\ &\textit{exset-rep}(< \mathbf{d}, \mathbf{r}, \text{split}, \text{tree}_{\textit{left}}, \text{tree}_{\textit{right}} >) &&= \\ &\qquad \textit{exset-rep}(\text{tree}_{\textit{left}}) \cup \{(\mathbf{d}, \mathbf{r})\} \cup \textit{exset-rep}(\text{tree}_{\textit{right}}) \end{aligned} \tag{6.4}$$

The invariant is that subtrees only ever contain **dom-elt**s which are on the correct side

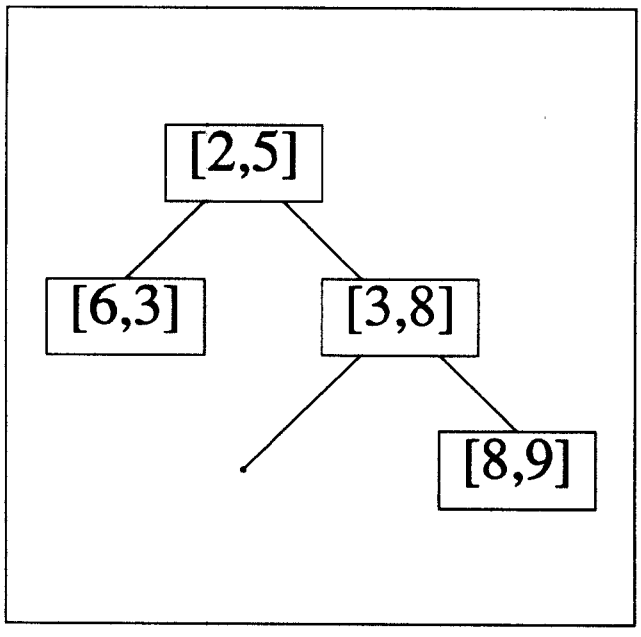

Figure 6.1

A 2d-tree of four elements. The splitting planes are not indicated. The [2,5] node splits along the y = 5 plane and the [3,8] node splits along the x = 3 plane.

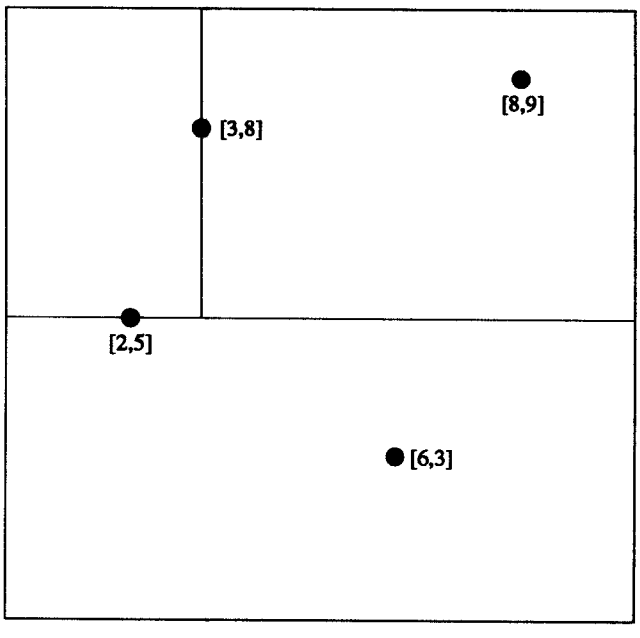

Figure 6.2

How the tree of Figure 6.1 splits up the x,y plane.

Algorithm:	Constructing a *k*d-tree
Input:	**exset**, of type **exemplar-set**
Output:	**kd**, of type **kdtree**
Pre:	None
Post:	**exset** = *exset-rep*(**kd**) $\wedge$ *Is-legal-kdtree*(**kd**)
Code:	
1.	If **exset** is empty then return the empty kdtree
2.	Call pivot-choosing procedure, which returns two values: **ex** := a member of **exset** **split** := the splitting dimension
3.	**d** := domain vector of **ex**
4.	**exset'** := **exset** with **ex** removed
5.	**r** := range vector of **ex**
6.	**exsetleft** := $\{(\mathbf{d}', \mathbf{r}') \in \mathbf{exset'} \mid d'_{\text{split}} \leq d_{\text{split}}\}$
7.	**exsetright** := $\{(\mathbf{d}', \mathbf{r}') \in \mathbf{exset'} \mid d'_{\text{split}} > d_{\text{split}}\}$
8.	**kdleft** := recursively construct *k*d-tree from **exsetleft**
9.	**kdright** := recursively construct *k*d-tree from **exsetright**
10.	**kd** := $< \mathbf{d}, \mathbf{r}, \mathbf{split}, \mathbf{kdleft}, \mathbf{kdright} >$
Proof:	By induction on the length of **exset** and the definitions of *exset-rep* and *Is-legal-kdtree*.

Table 6.3: Constructing a *k*d-tree from a set of exemplars.

of all their ancestors' splitting planes.

$$
\begin{aligned}
&\textit{Is-legal-kdtree}(\text{empty}).\\
&\textit{Is-legal-kdtree}(< \mathbf{d}, \mathbf{r}, -, \text{empty}, \text{empty} >).\\
&\textit{Is-legal-kdtree}(< \mathbf{d}, \mathbf{r}, \text{split}, \text{tree}_{\textit{left}}, \text{tree}_{\textit{right}} >) \Leftrightarrow\\
&\qquad \forall(\mathbf{d}', \mathbf{r}') \in \textit{exset-rep}(\text{tree}_{\textit{left}})\ d'_{\text{split}} \leq d_{\text{split}} \wedge\\
&\qquad \forall(\mathbf{d}', \mathbf{r}') \in \textit{exset-rep}(\text{tree}_{\textit{right}})\ d'_{\text{split}} > d_{\text{split}} \wedge\\
&\qquad\qquad \textit{Is-legal-kdtree}(\text{tree}_{\textit{left}}) \wedge\\
&\qquad\qquad \textit{Is-legal-kdtree}(\text{tree}_{\textit{right}})
\end{aligned}
\tag{6.5}
$$

6.3.2 Constructing a kd-tree

Given an exemplar-set **E**, a *k*d-tree can be constructed by the algorithm in Table 6.3. The pivot-choosing procedure of Step 2 inspects the set and chooses a "good" domain vector from this set to use as the tree's root. The discussion of how such a root is chosen is deferred to Section 6.7. Whichever exemplar is chosen as root will not affect the correctness of the *k*d-tree, though the tree's maximum depth and the shape of the hyperregions *will* be affected.

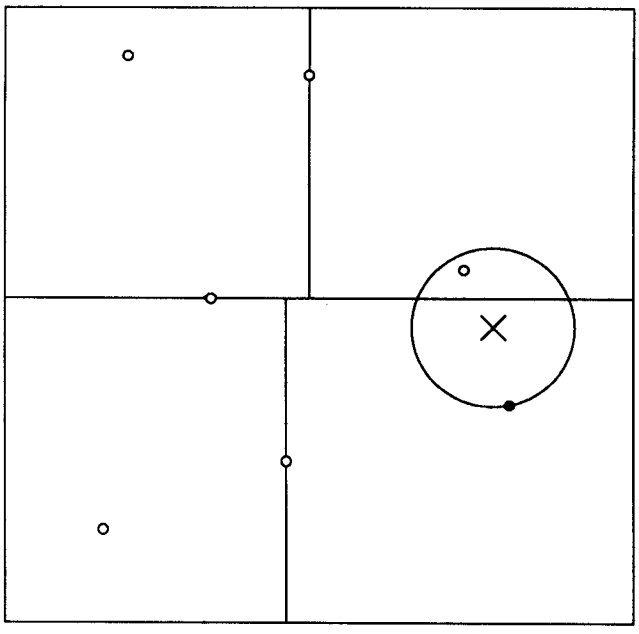

Figure 6.3

The black dot is the dot which owns the leaf node containing the target (the cross). Any nearer neighbour must lie inside this circle.

6.4 Nearest Neighbour Search

In this section, I describe the nearest neighbour algorithm which operates on *k*d-trees. I begin with an informal description and worked example, and then give the precise algorithm.

A first approximation is initially found at the leaf node which contains the target point. In Figure 6.3 the target point is marked X and the leaf node of the region containing the target is coloured black. As is exemplified in this case, this first approximation is not necessarily the nearest neighbour, but at least we know any potential nearer neighbour must lie closer, and so must lie within the circle centred on X and passing through the leaf node. We now back up to the parent of the current node. In Figure 6.4 this parent is the black node. We compute whether it is *possible* for a closer solution to that so far found to exist in this parent's other child. Here it is not possible, because the circle does not intersect with the (shaded) space occupied by the parent's other child. If no closer neighbour can exist in the other child, the algorithm can immediately move up a further level, else it must recursively explore the other child. In this example, the next parent which is checked will need to be explored, because the area it covers (i.e. everywhere north of the central horizontal line) *does* intersect with the best circle so far.

Table 6.4 describes my actual implementation of the nearest neighbour algorithm. It is called with four parameters: the *k*d-tree, the target domain vector, a representation of a hyperrectangle in **Domain**, and a value indicating the maximum distance from the

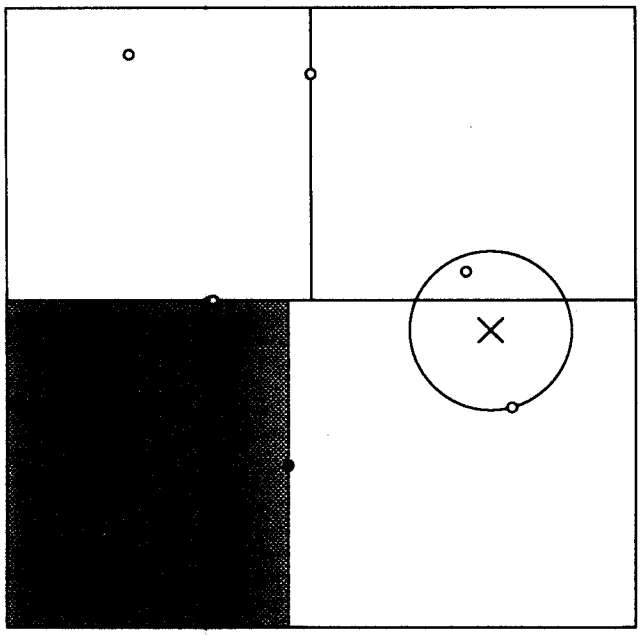

Figure 6.4

The black dot is the parent of the closest found so far. In this case the black dot's other child (shaded grey) need not be searched.

target which is worth searching. The search will only take place within those portions of the *k*d-tree which lie both in the hyperrectangle, and within the maximum distance to the target. The caller of the routine will generally specify the infinite hyperrectangle which covers the whole of **Domain**, and the infinite maximum distance.

Before discussing its execution, I will explain how the operations on the hyperrectangles can be implemented. A hyperrectangle is represented by two arrays: one of its minimum coordinates, the other of its maximum coordinates. To *cut* the hyperrectangle, so that one of its edges is moved closer to its centre, the appropriate array component is altered. To check to see if a hyperrectangle **hr** intersects with a hypersphere radius r centered at point **t**, we find the point **p** in **hr** which is closest to **t**. Write $hr_i^{\min}$ as the minimum extreme of **hr** in the ith dimension and $hr_i^{\max}$ as the maximum extreme. p_i, the ith component of this closest point is computed thus:

$$p_i = \begin{cases} hr_i^{\min} & \text{if } t_i \leq hr_i^{\min} \\ t_i & \text{if } hr_i^{\min} < t_i < hr_i^{\max} \\ hr_i^{\min} & \text{if } t_i \geq hr_i^{\max} \end{cases} \tag{6.6}$$

The objects intersect only if the distance between **p** and **t** is less than or equal to r.

The search is depth first, and uses the heuristic of searching first the child node which contains the target. Step 1 deals with the trivial empty tree case, and Steps 2 and 3 assign two important local variables. Step 4 cuts the current hyperrectangle into the two hyperrectangles covering the space occupied by the child nodes. Steps 5–7 determine

Algorithm:	Nearest Neighbour in a *k*d-tree
Input:	**kd**, of type **kdtree** **target**, of type **domain vector** **hr**, of type **hyperrectangle** **max-dist-sqd**, of type **float**
Output:	**nearest**, of type **exemplar** **dist-sqd**, of type **float**
Pre:	*Is-legal-kdtree*(**kd**)
Post:	Informally, the postcondition is that **nearest** is a nearest exemplar to **target** which also lies both within the hyperrectangle **hr** and within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. $\sqrt{\textbf{dist-sqd}}$ is the distance of this nearest point. If there is no such point then **dist-sqd** contains infinity.
Code:	
1.	if **kd** is empty then set **dist-sqd** to infinity and exit.
2.	**s** := **split** field of **kd**
3.	**pivot** := **dom-elt** field of **kd**
4.	Cut **hr** into two sub-hyperrectangles **left-hr** and **right-hr**. The cut plane is through **pivot** and perpendicular to the **s** dimension.
5.	**target-in-left** := $\textbf{target}_\textbf{s} \leq \textbf{pivot}_\textbf{s}$
6.	if **target-in-left** then
6.1	**nearer-kd** := **left** field of **kd** and **nearer-hr** := **left-hr**
6.2	**further-kd** := **right** field of **kd** and **further-hr** := **right-hr**
7.	if **not target-in-left** then
7.1	**nearer-kd** := **right** field of **kd** and **nearer-hr** := **right-hr**
7.2	**further-kd** := **left** field of **kd** and **further-hr** := **left-hr**
8.	Recursively call **Nearest Neighbour** with parameters (**nearer-kd**,**target**, **nearer-hr**,**max-dist-sqd**), storing the results in **nearest** and **dist-sqd**
9.	**max-dist-sqd** := minimum of **max-dist-sqd** and **dist-sqd**
10.	A nearer point could only lie in **further-kd** if there were some part of **further-hr** within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. if this is the case then
10.1	if $(\textbf{pivot} - \textbf{target})^2 < \textbf{dist-sqd}$ then
10.1.1	**nearest** := (**pivot**, **range-elt** field of **kd**)
10.1.2	**dist-sqd** := $(\textbf{pivot} - \textbf{target})^2$
10.1.3	**max-dist-sqd** := **dist-sqd**
10.2	Recursively call **Nearest Neighbour** with parameters (**further-kd**,**target**, **further-hr**,**max-dist-sqd**), storing the results in **temp-nearest** and **temp-dist-sqd**
10.3	If **temp-dist-sqd** < **dist-sqd** then
10.3.1	**nearest** := **temp-nearest** and **dist-sqd** := **temp-dist-sqd**
Proof:	Outlined in text

Table 6.4: The Nearest Neighbour Algorithm

which child contains the target. After Step 8, when this initial child is searched, it may be possible to prove that there cannot be any closer point in the hyperrectangle of the further child. In particular, the point at the current node must be out of range. The test is made in Steps 9 and 10. Step 9 restricts the maximum radius in which any possible closer point could lie, and then the test in Step 10 checks whether there is any space in the hyperrectangle of the further child which lies within this radius. If it is *not* possible then no further search is necessary. If it *is* possible, then Step 10.1 checks if the point associated with the current node of the tree is closer than the closest yet. Then, in Step 10.2, the further child is recursively searched. The maximum distance worth examining in this further search is the distance to the closest point yet discovered.

The proof that this will find the nearest neighbour within the constraints is by induction on the size of the *k*d-tree. If the cutoff were not made in Step 10, then the proof would be straightforward: the point returned is the closest out of (i) the closest point in the nearer child, (ii) the point at the current node and (iii) the closest point in the further child. If the cutoff *were* made in Step 10, then the point returned is the closest point in the nearest child, and we can show that neither the current point, nor any point in the further child can possibly be closer.

Many local optimizations are possible which while not altering the asymptotic performance of the algorithm will multiply the speed by a constant factor. In particular, it is in practice possible to hold almost all of the search state globally, instead of passing it as recursive parameters.

6.5 Theoretical Behaviour

Given a *k*d-tree with N nodes, how many nodes need to be inspected in order to find the proven nearest neighbour using the algorithm in Section 6.4?. It is clear at once that on average, at least $O(\log N)$ inspections are necessary, because any nearest neighbour search requires traversal to at least one leaf of the tree. It is also clear that no more than N nodes are searched: the algorithm visits each node at most once.

Figure 6.5 graphically shows why we might expect considerably fewer than N nodes to be visited: the shaded areas correspond to areas of the *k*d-tree which were cut off.

The important values are (i) the worst case number of inspections and (ii) the expected number of inspections. It is actually easy to construct worst case distributions of the points which will force nearly all the nodes to be inspected. In Figure 6.6, the tree is two-dimensional, and the points are scattered along the circumference of a circle. If we request the nearest neighbour with the target close to the centre of the circle, it will therefore be necessary for each rectangle, and hence each leaf, to be inspected (this is in order to ensure that there is no point lying inside the circle in any rectangle).

Calculation of the expected number of inspections is difficult, because the analysis depends critically on the expected distribution of the points in the *k*d-tree, and the expected

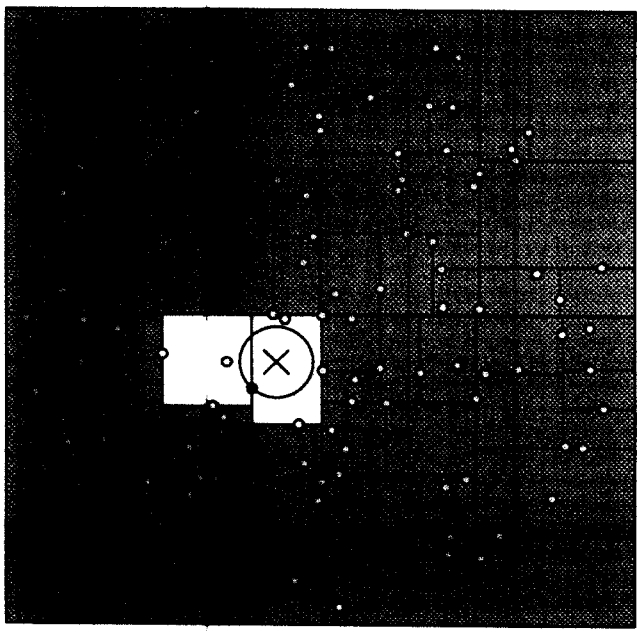

Figure 6.5

Generally during a nearest neighbour search only a few leaf nodes need to be inspected.

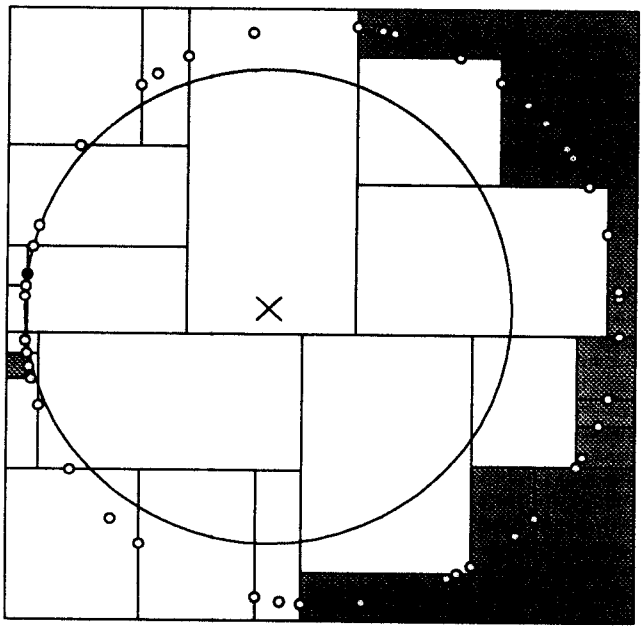

Figure 6.6

A bad distribution which forces almost all nodes to be inspected.

distribution of the target points presented to the nearest neighbour algorithm.

The analysis is performed in [Friedman *et al.*, 1977]. This paper considers the expected number of hyperrectangles corresponding to leaf nodes which will provably need to be searched. Such hyperrectangles intersect the volume enclosed by a hypersphere centered on the query point whose surface passes through the nearest neighbour. For example, in Figure 6.5 the hypersphere (in this case a circle) is shown, and the number of intersecting hyperrectangles is two.

The paper shows that the expected number of intersecting hyperrectangles is independent of N, the number of exemplars. The asymptotic search time is thus logarithmic because the time to descend from the root of the tree to the leaves is logarithmic (in a balanced tree), and then an expected constant amount of backtracking is required.

However, this reasoning was based on the assumption that the hyperrectangles in the tree tend to be hypercubic in shape. Empirical evidence in my investigations has shown that this is not generally the case for their tree building strategy. This is discussed and demonstrated in Section 6.7.

A second danger is that the cost, while independent of N, is exponentially dependent on k, the dimensionality of the domain vectors[1].

Thus theoretical analysis provides some insight into the cost, but here, empirical investigation will be used to examine the expense of nearest neighbour in practice.

6.6 Empirical Behaviour

In this section I investigate the empirical behaviour of the nearest neighbour searching algorithm. We expect that the number of nodes inspected in the tree varies according to the following properties of the tree:

- N, the size of the tree.
- k_{dom} , the dimensionality of the domain vectors in the tree. This value is the k in kd-tree.
- d_{distrib} , the distribution of the domain vectors. This can be quantified as the "true" dimensionality of the vectors. For example, if the vectors had three components, but all lay on the surface of a sphere, then the underlying dimensionality would be 2. In general, discovery of the underlying dimensionality of a given sample of points is extremely difficult, but for these tests it is a straightforward matter to *generate* such points. To make a kd-tree with underlying dimensionality d_{distrib}, I use randomly generated k_{dom}-dimensional domain vectors which lie on a d_{distrib}-dimensional hyperelliptical surface. The random vector generation algorithm is as follows: Generate d_{distrib} random angles $\theta_i \in [0, 2\pi)$ where $0 \leq i < d_{\text{distrib}}$. Then let

[1]This was pointed out to the author by N. Maclaren.

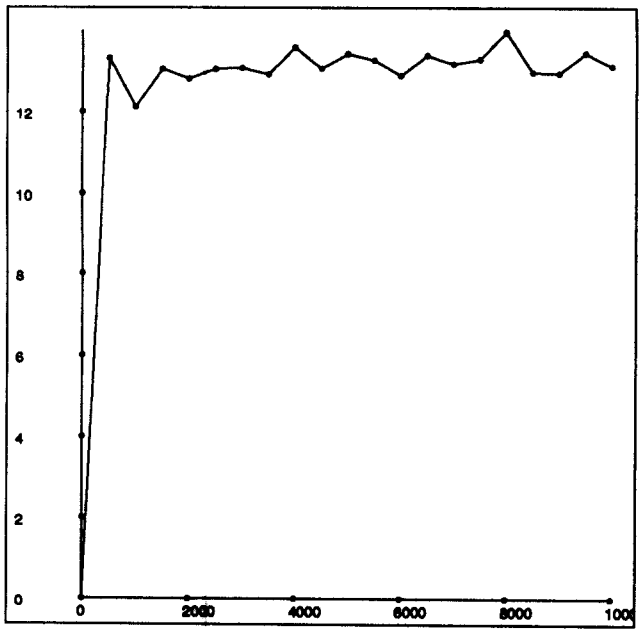

Figure 6.7

Number of inspections required during a nearest neighbour search against the size of the kd-tree. In this experiment the tree was four-dimensional and the underlying distribution of the points was three-dimensional.

the jth component of the vector be $\prod_{i=0}^{i=d-1} \sin(\theta_i + \phi_{ij})$. The phase angles ϕ_{ij} are defined as $\phi_{ij} = \frac{1}{2}\pi$ if the jth bit of the binary representation of i is 1 and is zero otherwise.

- d_{target} , the probability distribution from which the search target vector will be selected. I shall assume that this distribution is the same as that which determines the domain vectors. This is indeed what will happen when the *k*d-tree is used for learning control.

In the following sections I investigate how performance depends on each of these properties.

6.6.1 Performance against Tree Size

Figures 6.7 and 6.8 graph the number of nodes inspected against the number of nodes in the entire *k*d-tree. Each value was derived by generating a random *k*d-tree, and then requesting 500 random nearest neighbour searches on the *k*d-tree. The average number of inspected nodes was recorded. A node was counted as being inspected if the distance between it and the target was computed. Figure 6.7 was obtained from a 4d-tree with an distribution distribution $d_{\text{distrib}} = 3$. Figure 6.8 used an 8d-tree with an underlying distribution $d_{\text{distrib}} = 8$.

It is immediately clear that after a certain point, the expense of a nearest neighbour search has no detectable increase with the size of the tree. This agrees with the proposed

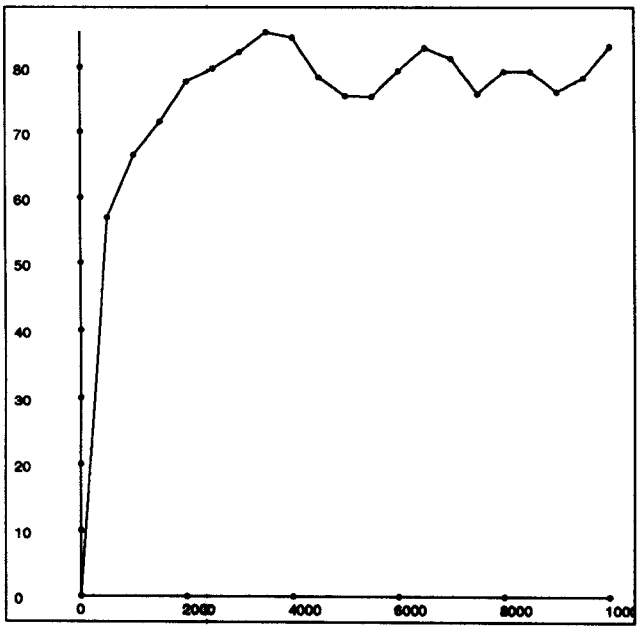

Figure 6.8

Number of inspections against kd-tree size for an eight-dimensional tree with an eight-dimensional underlying distribution.

model of search cost—logarithmic with a large additive constant term.

6.6.2 Performance against the "k" in kd-tree

Figure 6.9 graphs the number of nodes inspected against k_{dom}, the number of components in the *k*d-tree's domain vectors for a 10,000 node tree. The underlying dimensionality was also k_{dom}. The number of inspections per search rises very quickly, possibly exponentially, with k_{dom}. This behaviour, the massive increase in cost with dimension, is familiar in computational geometry.

6.6.3 Performance against the Distribution Dimensionality

This experiment confirms that it is d_{distrib}, the distribution dimension from which the points were selected, rather than k_{dom} which critically affects the search performance. The trials for Figure 6.10 used a 10,000 node *k*d-tree with domain dimension of 14, for various values of d_{distrib}. The important observation is that for 14d-trees, the performance does improve greatly if the underlying distribution-dimension is relatively low. Conversely, Figure 6.11 shows that for a fixed (4-d) underlying dimensionality, the search expense does not seem to increase any worse than linearly with k_{dom}.

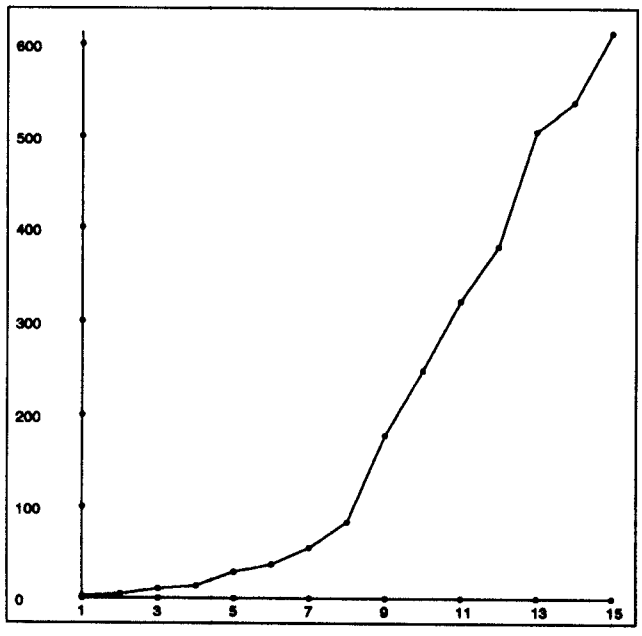

Figure 6.9

Number of inspections graphed against tree dimension. In these experiments the points had an underlying distribution with the same dimensionality as the tree.

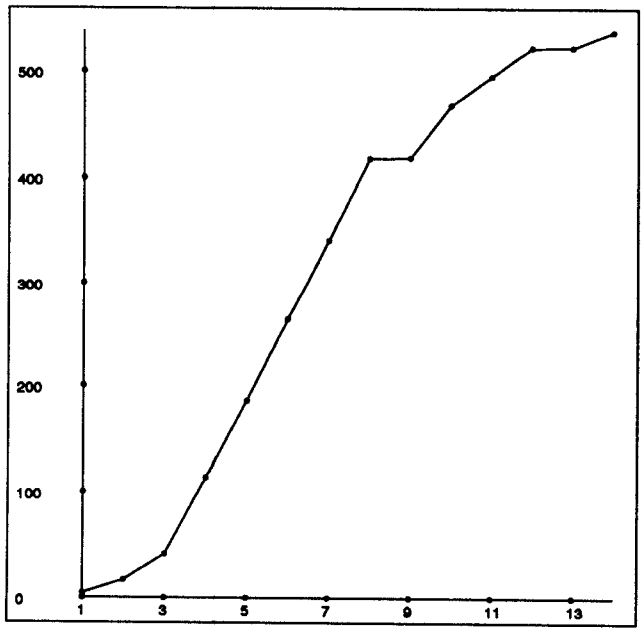

Figure 6.10

Number of inspections graphed against underlying dimensionality for a fourteen-dimensional tree.

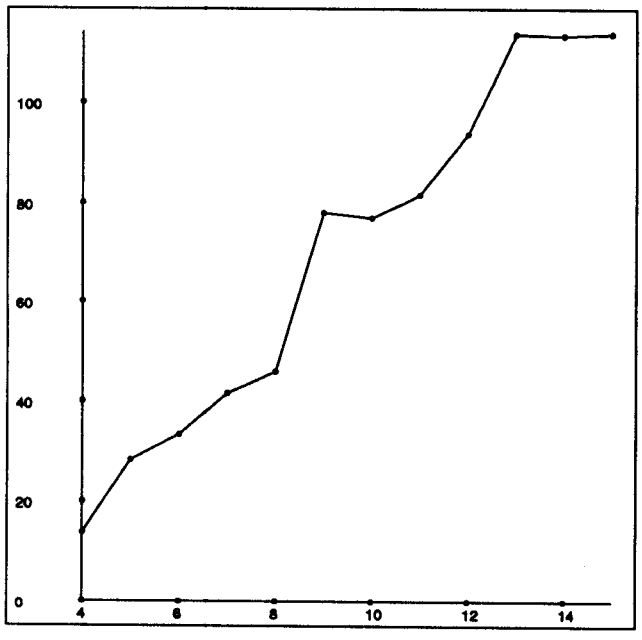

Figure 6.11

Number of inspections graphed against tree dimension, given a constant four dimensional underlying distribution.

6.6.4 When the Target is not Chosen from the kd-tree's Distribution

In this experiment the points were distributed on a three dimensional elliptical surface in ten-dimensional space. The target vector was, however, chosen at random from a ten-dimensional distribution. The *k*d-tree contained 10,000 points. The average number of inspections over 50 searches was found to be 8,396. This compares with another experiment in which both points and target were distributed in ten dimensions and the average number of inspections was only 248. The reason for the appalling performance was exemplified in Figure 6.6: if the target is very far from its nearest neighbour then very many leaf nodes must be checked.

6.6.5 Conclusion

The speed of the search (measured as the number of distance computations required) seems to vary ...

- ...only marginally with tree size. If the tree is sufficiently large with respect to the number of dimensions, it is essentially constant.
- ...very quickly with the dimensionality of the distribution of the datapoints, d_{distrib}.
- ...linearly with the number of components in the *k*d-tree's domain (k_{dom}), given a fixed distribution dimension (d_{distrib}).

There is also evidence to suggest that unless the target vector is drawn from the same distribution as the *k*d-tree points, performance can be greatly worsened.

These results support the belief that real time searching for nearest neighbours is practical in a robotic system where we can expect the underlying dimensionality of the data points to be low, roughly less than 10. This need not mean that the vectors in the input space should have less than ten components. For data points obtained from robotic systems it will not be easy to decide what the underlying dimensionality is. However Chapter 10 will show that the data does tend to lie within a number of low dimensional subspaces.

6.7 Further kd-tree Operations

In this section I discuss some other operations on *k*d-trees which are required for use in the *SAB* learning system. These include incrementally adding a point to a *k*d-tree, range searching, and selecting a pivot point.

6.7.1 Range Searching a kd-tree

$$\textit{range-search} \;:\; \text{exemplar-set} \times \mathbf{Domain} \times \Re \rightarrow \text{exemplar-set}$$

The abstract range search operation on an exemplar-set finds all exemplars whose domain vectors are within a given distance of a target point:

$$\textit{range-search}(\mathbf{E}, \mathbf{d}, r) \;=\; \{(\mathbf{d}', \mathbf{r}') \in \mathbf{E} \mid (\mathbf{d} - \mathbf{d}')^2 < r^2\}$$

This is implemented by a modified nearest neighbour search. The modifications are that (i) the initial distance is not reduced as closer points are discovered and (ii) all discovered points within the distance are returned, not just the nearest. The complexity of this operation is shown, in [Preparata and Shamos, 1985], to still be logarithmic in N (the size of $\mathbf{E}$) for a fixed range size.

6.7.2 Choosing a Pivot from an Exemplar Set

The tree building algorithm of Section 6.3 requires that a pivot and a splitting plane be selected from which to build the root of a *k*d-tree. It is desirable for the tree to be reasonably balanced, and also for the shapes of the hyperregions corresponding to leaf nodes to be fairly equally proportioned. The first criterion is important because a badly unbalanced tree would perhaps have $O(N)$ accessing behaviour instead of $O(\log N)$. The second criterion is in order to maximize cutoff opportunities for the nearest neighbour search. This is difficult to formalize, but can be motivated by an illustration. In Figure 6.12 is a perfectly balanced *k*d-tree in which the leaf regions are very non-square. Figure 6.13

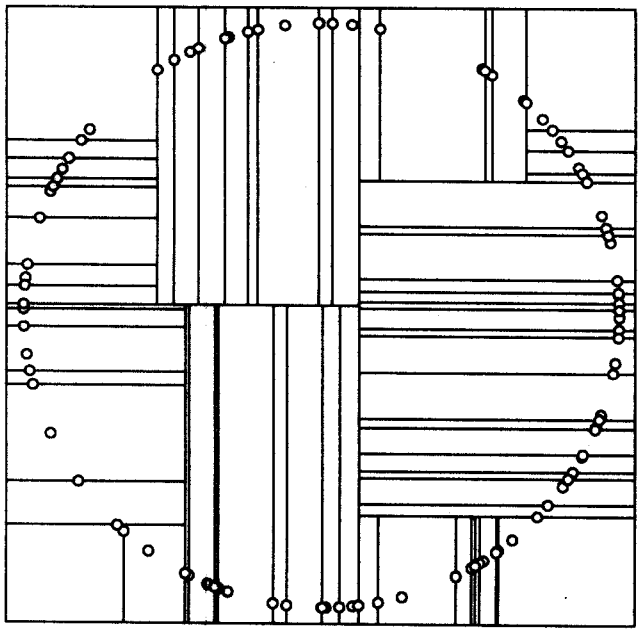

Figure 6.12

A 2d tree balanced using the 'median of the most spread dimension' pivoting strategy.

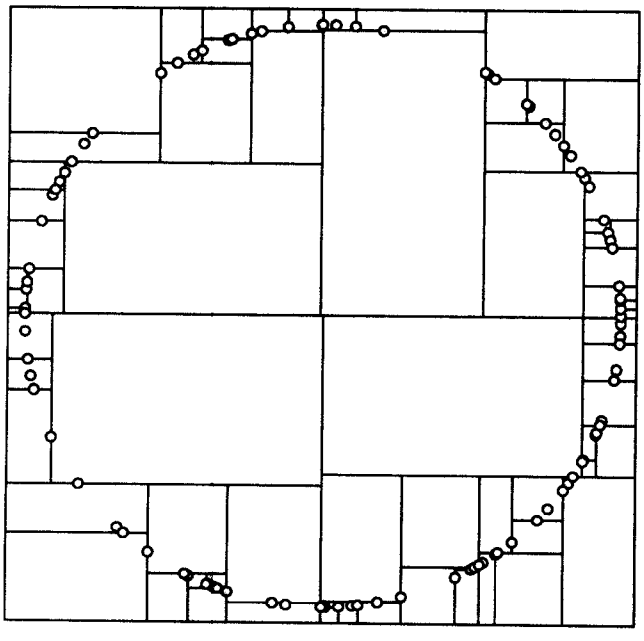

Figure 6.13

A 2d tree balanced using the 'closest to the centre of the widest dimension' pivoting strategy.

illustrates a *k*d-tree representing the same set of points, but which promotes squareness at the expense of some balance.

One pivoting strategy which would lead to a perfectly balanced tree, and which is suggested in [Omohundro, 1987], is to pick the splitting dimension as that with maximum variance, and let the pivot be the point with the median split component. This will, it is hoped, tend to promote square regions because having split in one dimension, the next level in the tree is unlikely to find that the same dimension has maximum spread, and so will choose a different dimension. For uniform distributions this tends to perform reasonably well, but for badly skewed distributions the hyperregions tend to take long thin shapes. This is exemplified in Figure 6.12 which has been balanced using this standard median pivot choice.

To avoid this bad case, I choose a pivot which splits the exemplar set in the middle of the range of the most spread dimension. As can be seen in Figure 6.13, this tends to favour squarer regions at the expense of a slight imbalance in the *k*d-tree. This means that large empty areas of space are filled with only a few hyperrectangles which are themselves large. Thus, the number of leaf nodes which need to be inspected in case they contain a nearer neighbour is smaller than for the original case, which had many small thin hyperrectangles.

My pivot choice algorithm is to firstly choose the splitting dimension as the longest dimension of the current hyperrectangle, and then choose the pivot as the point closest to the middle of the hyperrectangle along this dimension. Occasionally, this pivot may even be an extreme point along its dimension, leading to an entirely unbalanced node. This is worth it, because it creates a large empty leaf node. It is possible but extremely unlikely that the points could be distributed in such a way as to cause the tree to have one empty child at every level. This would be unacceptable, and so above a certain depth threshold, the pivots are chosen using the standard median technique.

Selecting the median as the split and selecting the closest to the centre of the range are both $O(N)$ operations, and so either way a tree rebalance is $O(N \log N)$.

6.7.3 Incrementally Adding a Point to a kd-tree

Firstly, the leaf node which contains the new point is computed. The hyperrectangle corresponding to this leaf is also obtained. See Section 6.4 for hyperrectangle implementation. When the leaf node is found it may either be (i) empty, in which case it is simply replaced by a new singleton node, or (ii) it contains a singleton node. In case (ii) the singleton node must be given a child, and so its previously irrelevant split field must be defined. The split field should be chosen to preserve the squareness of the new subhyperrectangles. A simple heuristic is used. The split dimension is chosen as the dimension in which the hyperrectangle is longest. This heuristic is motivated by the same requirement as for tree balancing—that the regions should be as square as possible, even if this means some loss of balance.

This splitting choice is just a heuristic, and there is no guarantee that a series of points added in this way will preserve the balance of the *k*d-tree, nor that the hyperrectangles will be well shaped for nearest neighbour search. Thus, on occasion (such as when the

depth exceeds a small multiple of the best possible depth) the tree is rebuilt. Incremental addition costs $O(\log N)$.

6.7.4 Q Nearest Neighbours

This uses a modified version of the nearest neighbour search. Instead of only searching within a sphere whose radius is the closest distance yet found, the search is within a sphere whose radius is the Qth closest yet found. Until Q points have been found, this distance is infinity.

6.7.5 Deleting a Point from a kd-tree

If the point is at a leaf, this is straightforward. Otherwise, it is difficult, because the structure of both trees below this node are pivoted around the point we wish to remove. One solution would be to rebuild the tree below the deleted point, but on occasion this would be very expensive. My solution is to mark the point as deleted with an extra field in the *k*d-tree node, and to ignore deletion nodes in nearest neighbour and similar searches. When the tree is next rebuilt, all deletion nodes are removed.

Chapter 7

SAB-trees: Coping with Disorder

The nearest neighbour (see Chapter 5) has been selected for learning world models which relate "state", "action" and "behaviour" (see Chapter 4). The previous chapter explained in detail how the nearest neighbour can be found efficiently. This chapter explains how the standard nearest neighbour algorithm is tailored to the needs of this investigation. It begins by explaining how kd-trees are used. It then shows how resistance to noise is implemented in a way which means that each access still only requires a single nearest neighbour search. After that, the problem of adapting to a changing world model is addressed. The next section explains the detailed implementation and asymptotic efficiency of these operations. The final section describes garbage collection of the kd-tree, to remove old, inaccurate, exemplars.

7.1 SAB Relations, and their Representation

The Perceived State Transition Function has this form:

$$\text{PSTF} : \textbf{State} \times \textbf{Action} \rightarrow \textbf{Behaviour} \tag{7.1}$$

All experiences are stored explicitly in a *k*d-tree. The choice of which components of the PSTF form the tree's domain (i.e. index), and which form the range is influenced by the fields which need a nearest neighbour search. Chapter 5 described how both *Dynamics Prediction*, searching on **State** × **Action**, and *Partial Inversion*, searching on **State** × **Behaviour**, would be useful.

The two different indexing requirements could be satisfied by various means. One method would be to index the *k*d-tree on all three components. During the search, when a split is found in a dimension which is *not* being used as an index, then both subtrees are searched. This is slower than if the search had been on a tree only indexed with the relevant components, but it is nevertheless faster than a complete linear search.

The solution to multiple indexes which I use is simpler. There are two *k*d-trees, one indexed by **State**×**Action** and the other by **State**×**Behaviour**. This uses more memory

but is faster, because nearest neighbour search takes place in a lower dimension. The extra memory is significantly less than twice as much, because the second tree can have pointers to the vectors of the first tree, rather than its own copy of each **s**, **b** and **a** vector.

7.2 Resisting Noise

Each exemplar is obtained by observing a state, applying an action and then observing the behaviour. The assumption that the sensor readings are sufficient to determine the state implies that if the system is ever in the same perceived state again and the same raw action is applied then the same behaviour will be observed. More simply we assume that the PSTF is a deterministic function. A problem is that non-deterministic random noise is likely to corrupt any observations or raw action signals. This means that on different occasions, the same state-action pair may produce different behaviour.

Let $\mathbf{c} \in \mathbf{State} \times \mathbf{Action}$ be a state-action pair which the controller believes it has experienced. In fact this belief could be wrong for two reasons:

1. The state observation could have an error.
2. The action signal which the controller requested could have been corrupted before it arrived at the actuator.

Imagine that the actual state action pair had been $\mathbf{c} + \boldsymbol{\delta c}$. Let $\mathbf{b}$ be the behaviour which then occurred. Due to noisy sensors the controller observes behaviour $\mathbf{b} + \boldsymbol{\delta b}$. Then the world model is updated with the observation that $\mathbf{c} \rightarrow \mathbf{b} + \boldsymbol{\delta b}$ when in fact the truth is that

$$\mathrm{PSTF}(\mathbf{c} + \boldsymbol{\delta c}) = \mathbf{b} \tag{7.2}$$

By the Taylor expansion

$$\mathrm{PSTF}(\mathbf{c} + \boldsymbol{\delta c}) = \mathrm{PSTF}(\mathbf{c}) + J\boldsymbol{\delta c} + O(\delta^2) \text{ where } J_{ij} = \frac{\partial \mathrm{PSTF}_i}{\partial c_j} \tag{7.3}$$

where J is the Jacobian matrix of the PSTF at $\mathbf{c}$. If $|\,\boldsymbol{\delta b}\,|$ and $|\,\boldsymbol{\delta c}\,|$ are sufficiently small the $O(\delta^2)$ term can be discarded. Thus the controller has wrongly recorded

$$\mathbf{c} \rightarrow \mathrm{PSTF}(\mathbf{c}) + \underbrace{J\boldsymbol{\delta c} + \boldsymbol{\delta b}}_{\text{error}} \tag{7.4}$$

The solution to the problem is to use the local average value for prediction. If we make a series of N observations at what the controller believes to be constant state-action $\mathbf{c}$ and then average the results we can hope to reduce the error. Let the ith observation have errors $\boldsymbol{\delta b}_i$ and $\boldsymbol{\delta c}_i$. Then the average estimate for $\mathrm{PSTF}(\mathbf{c})$ is

$$\frac{1}{N}\sum_{i=1}^{N}(\mathbf{b} + \boldsymbol{\delta b}_i). \tag{7.5}$$

$$= \frac{1}{N}\sum_{i=1}^{N}(\text{PSTF}(\mathbf{c}) + J\delta\boldsymbol{c}_i + \delta\boldsymbol{b}_i). \tag{7.6}$$

$$= \text{PSTF}(\mathbf{c}) + J\left(\frac{1}{N}\sum_{i=1}^{N}\delta\boldsymbol{c}_i\right) + \frac{1}{N}\sum_{i=1}^{N}\delta\boldsymbol{b}_i. \tag{7.7}$$

If the $\delta\boldsymbol{b}_i$s and $\delta\boldsymbol{c}_i$s are both drawn from random distributions with means $\mathbf{m}_b$ and $\mathbf{m}_c$ then as N increases the value converges to

$$\text{PSTF}(\mathbf{c}) + J\mathbf{m}_c + \mathbf{m}_b. \tag{7.8}$$

If these means are zero then the estimate converges to the correct value as required. What would it imply if the error distributions did not have zero mean? An example of this in the Mountain Car domain would be where the pedal on average was pushed 10mm further down than the controller requested. As a result, the speed obtained by requesting a pedal of 80mm for example would be predicted wrongly as that obtained from an actual pedal value of 70mm. But is this wrong? In fact it is not, because of the fundamental fact that *we are trying to learn the relationship between what we request and what we perceive* rather than what the actual control was and what the actual result was. Another illustration is for supposed regular errors in perception. If a vision controlled arm regularly thinks things on the retina are an inch to the left of where they "should" be (for example due to the camera having been set up slightly askew) and then is shown a goal position to move to, it will then try to move the "wrong" image of the hand to the "wrong" image of the goal and will thus succeed. There will only be a problem if the task is specified in a different coordinate system to the state description. In this case a regular error would necessarily fool any controller using any form of sensory feedback. This seems reasonable: a person performing a reaching task should still manage it wearing distorting glasses, but would find it very hard to cope if the goal's apparent position were distorted but the hand not.

The mean errors are zero and so the averaging estimate converges in the presence of noise to the correct value. The rate of convergence as learning progresses will depend on the amount of noise (the variance of the noise distributions) and the number of experiences of the state-action in question. The important state-actions which are frequently needed will converge more quickly than those which are rarely used (and these are state-action pairs for which less accuracy is likely to be needed).

I will now describe in more detail how local averaging is used in this *SAB* implementation. It takes place over all points within a certain distance D_{range} (the *range distance*) of the query point. This speeds the convergence because there are more data points to examine. Using spatially local points for averaging is analogous to a common image processing operation, in which image noise is removed by replacing all pixels by the local weighted average of pixel values. The disadvantage is the need to choose the range distance D_{range}. Too small a distance will provide little averaging of the noisy values. Too large a distance

will smooth too harshly and genuine details in the underlying function will be lost. The system designer must choose the range distance based on the expected noise and expected smoothness of the function. In the experimental studies this has not been a difficult choice and the behaviour has not been sensitive to variation in the range width. If necessary, it is possible that the range width could be obtained automatically by more sophisticated statistical analysis or a cross-validation technique.

The weight function gives more weight to points which lie close to the query point. The smoothed prediction from a set of exemplars

$$\mathbf{E} = \{(\mathbf{s}_1, \mathbf{a}_1, \mathbf{b}_1), \ldots, (\mathbf{s}_N, \mathbf{a}_N, \mathbf{b}_N)\} \tag{7.9}$$

is

$$eval_{\mathbf{E}}(\mathbf{s}, \mathbf{a}) = \frac{\sum_{i=1}^{N} (weight(dist_i) \times \mathbf{b}_i)}{\sum_{i=1}^{N} weight(dist_i)} \tag{7.10}$$

where $dist_i = |\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}, \mathbf{a})\ |$ and the weight strength decreases to zero beyond the range distance $D_{\mathbf{range}}$ according to the bisquare function

$$weight(d) = \begin{cases} \left[1 - (d/D_{\mathbf{range}})^2\right]^2 & \text{if } d < D_{\mathbf{range}} \\ 0 & \text{if } d \geq D_{\mathbf{range}} \end{cases} \tag{7.11}$$

It is graphed in Figure 7.1.

In Section 7.4 I explain how $eval_{\mathbf{E}}(\mathbf{s}, \mathbf{a})$ is computed with adequate efficiency. At this point I note simply that to avoid having to perform this local averaging every time a query occurs, the smoothed value at each exemplar is actually stored with the exemplar as a fourth component $\mathbf{b}^{\mathbf{smooth}}$:

$$\mathbf{E} = \{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i, \mathbf{b}_i^{\mathbf{smooth}}) : \mathbf{b}_i^{\mathbf{smooth}} = eval_{\mathbf{E}}(\mathbf{s}_i, \mathbf{a}_i)\} \tag{7.12}$$

When a prediction is required for a point which has not yet been experienced, the smoothed prediction is not computed. Instead the $\mathbf{b}^{\mathbf{smooth}}$ component of the nearest neighbour is used. As well as reducing computation this has an important property for sparsely populated regions. The prediction will be more useful than the "zero" prediction provided by $eval_{\mathbf{E}}$ when there are no exemplars within the range distance. The $\mathbf{b}^{\mathbf{smooth}}$ component of an exemplar is initially computed when the exemplar is added to the *SAB*-tree. It is updated only when necessary to preserve the invariant $\mathbf{b}_i^{\mathbf{smooth}} = eval_{\mathbf{E}}(\mathbf{s}_i, \mathbf{a}_i)$.

This method has been introduced to reduce the effects of noisy exemplars. *It is not intended to, nor expected to, provide extra interpolative accuracy.* Here is an illustrative example. Suppose **S** is a zero-dimensional space, **A** is one-dimensional and **B** is one-dimensional and suppose that we have observed the noisy exemplars which are the solid dots in Figure 7.2. The white dots are the smoothed value at each point. The line shows

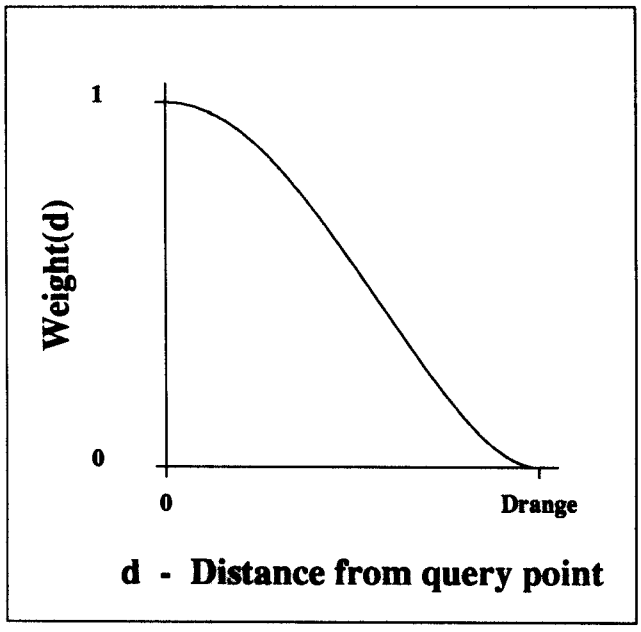

Figure 7.1

The smoothing function. The weight of a near neighbour decreases with distance to the query point. It has no influence beyond distance D_{range}.

the function produced by using the smoothed nearest neighbour. Figure 7.3 shows what happens if too small a value of D_{range} is used (equivalent to not using smoothing), and Figure 7.4 shows what happens if too large a value is used. Visually, the smoothing appears adequate for values of D_{range} between 5% and 40% of the diagram width.

Finally, Figures 7.5 and 7.6 show the Mountain Car PSTF respectively not using and using a smoothing kernel in the presence of simulated noise.

Before concluding this section it is worth noting another interpretation of the $eval_{\mathbf{E}}$ smoothing operation. It is an example of Shepards Interpolation [Franke, 1982]. Our analogous operation, averaging, is performed locally. We have a tighter restriction on the shape of the smoothing function than would a user of Shepard's Interpolation: we require it reach zero within a relatively short range of the query point: Shepard's interpolation weight functions are usually non-zero over the entire domain.

7.3 Adapting to a Changing Environment

As well as disorder in the form of noise there is another problem that a learning controller should be able to cope with: an environment that changes unpredictably with time. Here are some examples of how this might come about:

1. **Gradual Change.** A new robot arm is put to work on a job. As time passes the stiff joints become less stiff due to wear, and so gradually over the first week of use the joint friction is reduced, changing the system dynamics.

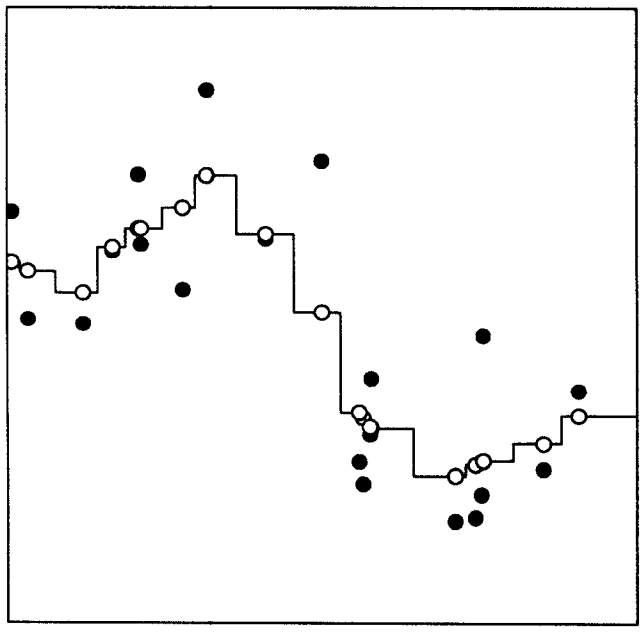

Figure 7.2

Learning from noisy exemplars. The y-coords of the black dots are the $\mathbf{b}$ components; those of the white dots are the $\mathbf{b}^{\mathbf{smooth}}$ components. The line is the interpretation induced by nearest neighbour. $D_{\mathbf{range}}$ is 10% of the diagram width.

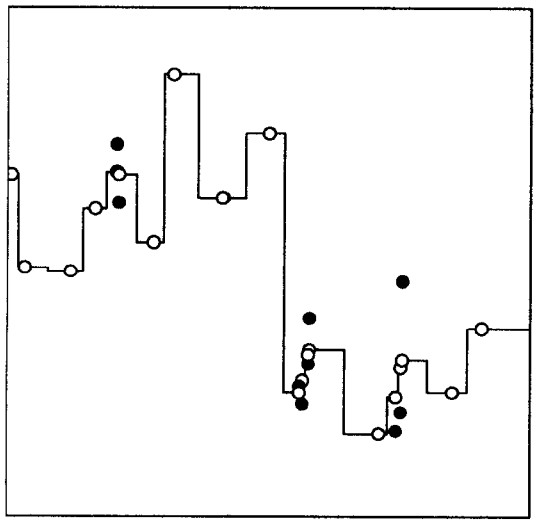

Figure 7.3: As Figure 7.2 except that $D_{\mathbf{range}}$ is 2% of the diagram width.

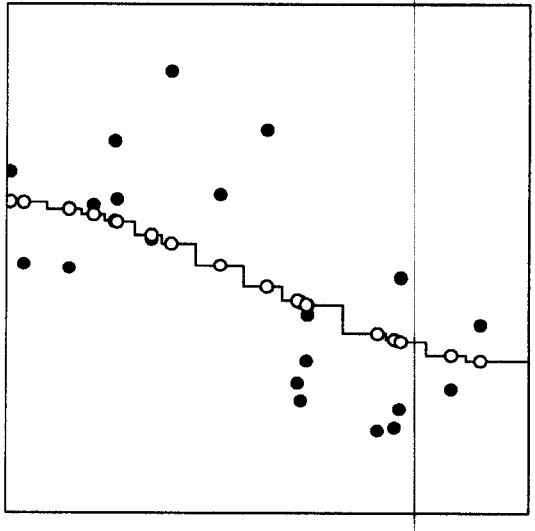

Figure 7.4: As Figure 7.2 except that $D_{\mathbf{range}}$ is 60% of the diagram width.

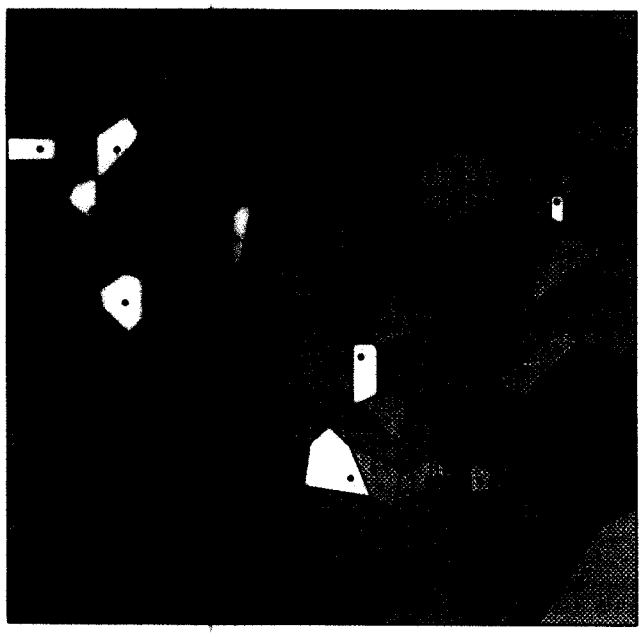

Figure 7.5

The mountain car PSTF (see Section 4.3). Noise of 50% was added to the perceived behaviour. This is the nearest neighbour interpretation using the $\mathbf{b}$ components of exemplars.

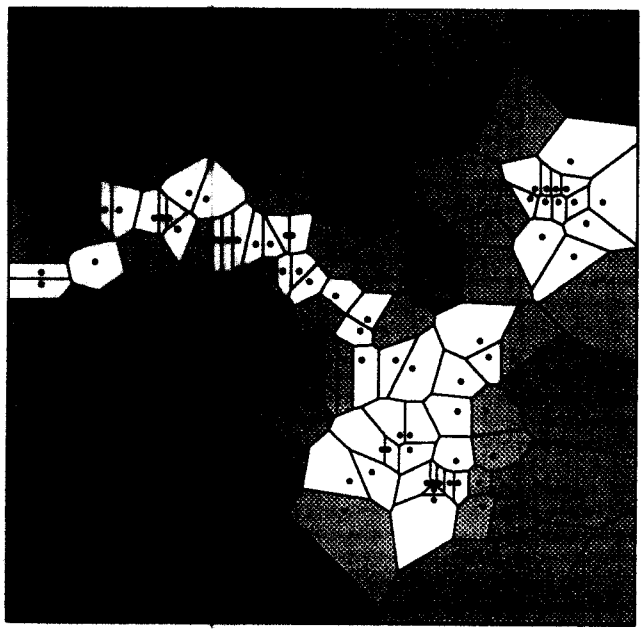

Figure 7.6

As Figure 7.5, but using the $\mathbf{b}^{\text{smooth}}$ components of exemplars. The range width, D_{range}, was 1.5 units.

2. **Sudden Change.** A vision-controlled robot is picking objects from a conveyor belt. After several months of operation the camera is jogged and instantly all observations are perceived to be two inches to left of their original locations.

An important feature of these changes is that they occur unpredictably. They could not in principle have been modelled, nor could their occurrence have even been announced.

Adjusting to this sort of event is an important problem for any learning controller, but for the nearest-neighbour learning described so far, it is particularly serious. The old, wrong exemplars will never disappear. During a gradual change, the old exemplars will initially be more useful than no knowledge at all, because their prediction will be close to correct though not perfect. But after a time it would have been better to be able to ignore the old exemplars. After a sudden change the old exemplars will all be misleading in the region of control space in which the change has occurred.

One possible solution would be to try to spot when a change had occurred by monitoring the recent history of errors. When this exceeded a threshold the controller would experience enforced amnesia, forgetting everything, and then begin relearning. This extreme approach would have two disadvantages. After a fairly small change (or a short period of a gradual change) the old, slightly wrong knowledge that the controller has is still more useful than no knowledge at all. Furthermore we should hope to be able to use the imperfect model to help guide the search for new, accurate knowledge. The second reason that it would be unfortunate to throw away all the knowledge is that there might be cases where the change only badly affects particular regions of the control space.

Instead, we are able to detect when it would be useful to throw away old knowledge, and when it is still useful to retain it. This is by comparing suspect old exemplars with their local neighbours. Because of the $\mathbf{b}^{\text{smooth}}$ modification to *SAB*-trees, this can be done reasonably cheaply.

Whenever a new point is added, all its local neighbours (within the range width, D_{range}, of Section 7.2) are inspected. An exemplar is deemed undesirable if...

- ...it disagrees with the local consensus of the value of the behaviour in its neighbourhood. For suspect exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i, \mathbf{b}_i^{\text{smooth}})$ this can be detected by noticing a large difference between $\mathbf{b}_i$ and $\mathbf{b}_i^{\text{smooth}}$. The test is whether $| \mathbf{b}_i - \mathbf{b}_i^{\text{smooth}} | > \Delta_{\text{old}}$ where Δ_{old} is the tolerable difference threshold.

- ...and it is relatively old. This is detected by storing the date of birth of each exemplar, and counting an exemplar as old if it is older than the mean age of the local exemplars.

Old, inaccurate points are simply deleted from the *SAB*-tree.

Examples of adapting to change for a noisy, one-dimensional mapping can be seen in Figure 7.7.

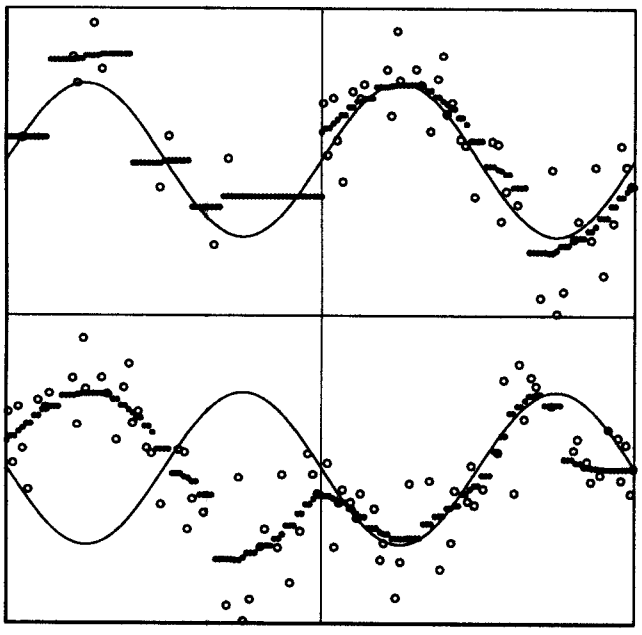

Figure 7.7

One-dimensional *SAB*-tree, attempting to learn a sine wave from noisy exemplars, shown as white dots. Black dots are the smoothed interpretation. Top left: After 8 random exemplars. Top right: After 50. Bottom Left: The underlying function is changed. Bottom right: After further 50 exemplars.

7.3.1 Discussion

Empirically, this adaptation mechanism is seen to work sufficiently well (c.f. Chapter 10). However, several improvements are possible, based on more detailed statistical analysis of the local points, for example by comparing the variance of the behaviours of (i) all local exemplars and (ii) recent local exemplars. This would help in the case where the exemplars are noisy, and the old exemplars' inaccuracies are not because of environmental changes. It is not clear that the avoidance of these occasional unjust exemplar removals is sufficient justification for the extra computation and hand-crafted statistical thresholds which would be required.

7.4 Updating the Relation

This section details the computation and the cost of the smoothing and adapting operations of the previous sections. They are implemented using the algorithms described in Chapter 6. The reader who is content to simply understand the behaviour without reading about the details may omit the rest of this chapter with no impact on later sections.

7.4.1 Duplicate Points

An additional, rather uninteresting, optimization occurs in the case when a new exemplar $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})$ is to be stored, but it is noticed that there is already an exemplar

$(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i)$ in the set $\mathbf{E}$ which is close enough in all three components to be considered identical. An example of where this might happen is when a robot arm is being successfully held stationary by *SAB* control. In this case the new point is not added, but the *mass* field of the exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i)$ is increased by one. The mass field is a small integer associated with each exemplar, which is initially one when a new exemplar is added to the tree. The mass field is necessary to preserve the definition of the $eval_{\mathbf{E}}$ operation, which now takes the mass of the exemplars into account in the local averaging operation (as in Equation 7.14 below). When an exemplar's mass is increased its date of birth is set to the current date (it is born again).

The obvious motivation for this optimization is that there is some saving in memory for very little extra work (provided that the memory for all the exemplars not duplicated exceeds the extra memory to include a mass field in each exemplar). However the main reason is that duplicate points stored in a *k*d-tree hinder nearest neighbour search which would inevitably have to search all of them if it would need to search one.

The mass feature is not included to alter the meaning of the exemplar set representation—it is a modification to the representation which preserves meaning.

7.4.2 *SAB*-tree Representation: Details

An *SAB*-tree has two roots: an *SA*-tree root and an *SB*-tree root. The former is a *k*d-tree indexed on the $(\mathbf{s}, \mathbf{a})$ components, and the latter indexed on the $(\mathbf{s}, \mathbf{b})$ components. All the data is held in the nodes of the *SA*-tree, while the nodes of the *SB*-tree are simply pointers to *SA*-tree nodes which contain the same exemplar information. The difference between these *k*d-trees is that the hyperplanes splitting up the *SA*-tree space are in **State** × **Action** space, while the hyperplanes for the *SB*-tree are in **State** × **Behaviour** space. Thus, nearest neighbour (and related) searches can take place in either the $(\mathbf{s}, \mathbf{a})$ or $(\mathbf{s}, \mathbf{b})$ components by using the appropriate root. In either case the result is an *SA*-tree node.

Each node of the *SA*-tree contains the data shown in Table 7.1. The set of nodes are subject to the following invariants, which are maintained whenever a new exemplar is added:

$$W_i = \sum_{j \in locals(i)} (weight(|\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j)\ |) \times M_j) \tag{7.13}$$

$$\mathbf{b}_i^{\text{smooth}} = \frac{\sum_{j \in locals(i)} (weight(|\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j)\ |) \times M_i \times \mathbf{b}_j)}{W_i} \tag{7.14}$$

where $j \in locals(i)$ is true if $|\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j)\ | < D_{\text{range}}$ and the function $weight : \Re \rightarrow \Re$ is defined in Section 7.2.

$\mathbf{s}_i$	A vector of k_s floating point numbers
$\mathbf{a}_i$	A vector of k_a floating point numbers
$\mathbf{b}_i$	A vector of k_b floating point numbers
$\mathbf{b}_i^{\text{smooth}}$	A vector of k_b floating point numbers
T_i^{birth}	An integer: The number of exemplars which had been stored in the tree before this one was added
M_i	A short integer: the mass of this exemplar
W_i	A floating point number: the denominator of the local average calculation of $\mathbf{b}_i^{\text{smooth}}$, discussed below

Table 7.1: The fields of a *SA*-tree node

Operation	Action	Order of Cost
find-locals	Find all local exemplars in range of smoothing kernel.	$k_{sa}(S \log N)$
maintain-bsmooth	Evaluate $\mathbf{b}_{n+1}^{\text{smooth}}$.	$k_{sa}S$
	Update $\mathbf{b}_i^{\text{smooth}}$ components of locals.	$k_{sa}S$
remove-bad	Check for elderly and inaccurate locals.	$k_{sa}S$
	Delete bad points.	(see text)
insert-new	Find insertion tree nodes.	$(k_{sa} + k_{sb}) \log N$

Table 7.2: When a new exemplar is added

7.4.3 Update

All the computation is performed when a point is added to the *SAB*-tree instead of when the *SAB*-tree is queried to make a prediction. This is useful because there will be a maximum of one update per control cycle, whereas there may be many predictions required in order to assess different possible actions. Table 7.2 illustrates the sequence of operations when a new point $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})$ is added. In this table, N is the number of exemplars in the *SAB*-tree and S is the number of exemplars found to lie within the smoothing range of $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1})$. $k_{sa} = k_s + k_a$ and $k_{sb} = k_s + k_b$.

The **find-locals** operation makes a range search (also described in Section 6.7) of the *SAB*-tree in the $(\mathbf{s}, \mathbf{a})$ components, finding all exemplars within distance D_{range} of $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1})$. Assume that S are found. It is hard to predict in advance how the expected value of S will depend on the dimensionality of the data and the size of the *SAB*-tree, and so this will be left to empirical observation.

maintain-bsmooth evaluates the $\mathbf{b}^{\text{smooth}}$ of the new exemplar and simultaneously updates the locals. The smoothed values of the locals could be readjusted naively by

recursively recomputing the smoothed values for each local. This would involve a range search for each of the S locals, which would be expensive. Instead, the W_i field can be used to reduce the computation to $2k_b$ arithmetic operations for each local.

Let $\left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{old}}$ be the smoothed value of the ith local exemplar before the new exemplar, $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{b}_{n+1})$, is added. Let $\left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{new}}$ be the smoothed value after. Finally, let $\Delta W = \mathit{weight}(|\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_{n+1}, \mathbf{a}_{n+1})\ |)$. Then from Equation 7.14

$$\left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{old}} = \frac{\sum_{j \in \mathit{locals}(i)} (\mathit{weight}(|\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j)\ |) \times M_j \times \mathbf{b}_j)}{W_i} \tag{7.15}$$

and

$$\left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{new}} = \frac{\Delta W \times \mathbf{b}_{n+1} + \sum_{j \in \mathit{locals}(i)} (\mathit{weight}(|\ (\mathbf{s}_i, \mathbf{a}_i) - (\mathbf{s}_j, \mathbf{a}_j)\ |) \times M_j \times \mathbf{b}_j)}{\Delta W + W_i} \tag{7.16}$$

$$= \frac{\Delta W \times \mathbf{b}_{n+1} + \left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{old}} \times W_i}{\Delta W + W_i} \tag{7.17}$$

When $\left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{new}}$ has been updated, W_i is then updated to $W_i + \Delta W$. The algorithm to simultaneously update locals and find $\mathbf{b}_{n+1}{}^{\text{smooth}}$ is given in Table 7.3.

The **remove-bad** operation searches those local points which are older than the mean age and finds any which are deemed inaccurate by the measure of Section 7.3. These are deleted from the *SAB*-tree, which is achieved by simply marking the exemplar to be ignored in all future searches. Section 7.5 details how these dead nodes are occasionally garbage collected. When a point is deleted, all of its local neighbours must change their $\mathbf{b}^{\text{smooth}}$ components to take account of the deletion. The local neighbours of the exemplar to be deleted are found by a range search, and their $\mathbf{b}_i^{\text{smooth}}$ and W_i components are updated accordingly (by an algorithm analogous to the **maintain-bsmooth** algorithm).

The **insert-new** operation finds the correct node in each of the *SA*-tree and the *SB*-tree in which to store the new exemplar, using the incremental insertion operation from Section 6.7. During the search for the correct node it might find a near identical node, in which case it is not stored explicitly but instead implicitly, by increasing the mass of the near identical node.

7.5 SAB-tree Garbage Collection

The world model represented by a *SAB*-tree is updated and accessed *during* task execution. The local updates have the ideal abstract properties which allow us to consider the behaviour of the *SAB*-tree as a simple set of exemplars which can be accessed by nearest neighbour. When the task is not executing, there may be some desirable alterations which could be made to the *SAB*-tree. These are not intended to alter the meaning of it in any way, but to improve the expected access time for future nearest neighbour searches.

Algorithm:	Maintaining $\mathbf{b}_i^{\text{smooth}}$ components
Input:	$\mathbf{s}_{n+1}$, the new state vector
	$\mathbf{a}_{n+1}$, the new action vector
	$\mathbf{b}_{n+1}$, the new behaviour vector
	locals, the set of local vectors
Output:	$\mathbf{b}_{n+1}^{\text{smooth}}$, the new smoothed vector
	W_{n+1} the new denominator value
Changed:	The $\mathbf{b}_i^{\text{smooth}}$ and W_i components of each exemplar in **locals**
Preconditions:	Invariants of Equations 7.14 and 7.13
Postconditions:	$\mathbf{b}_{n+1}^{\text{smooth}}$, W_{n+1}, and the $\mathbf{b}_i^{\text{smooth}}$ component and W_i component of each exemplar in **locals** is updated to maintain the invariant upon the addition of the new exemplar.
Code:	
1.	**weighted-sum** $:= \mathbf{0}$
2.	$W_{n+1} := 0$
3.	for each exemplar $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{b}_i, \mathbf{b}_i^{\text{smooth}}, M_i, W_i) \in$ **locals**
3.1	$\Delta W := \mathit{weight}(\mid (\mathbf{s}_{n+1}, \mathbf{a}_{n+1}) - (\mathbf{s}_i, \mathbf{a}_i) \mid)$
3.2	$\mathbf{b}_i^{\text{smooth}} := \dfrac{\left[\mathbf{b}_i^{\text{smooth}}\right]_{\text{old}} \times W_i + \mathbf{b}_{n+1} \times \Delta W}{W_i + \Delta W}$
3.3	$W_i := W_i + \Delta W$
3.4	**weighted-sum** $:=$ **weighted-sum** $+ \Delta W \times M_i \times \mathbf{b}_i$
3.5	$W_{n+1} := W_{n+1} + \Delta W \times M_i$
4.	$\mathbf{b}_{n+1}^{\text{smooth}} := \dfrac{\textbf{weighted-sum}}{W_{n+1}}$

Table 7.3: Algorithm: The **maintain-bsmooth** operation. This evaluates the new smoothed behaviour component and adjusts the local smoothed components.

One exemplar set can be represented by many different SAB-trees, and some SAB-trees are more efficient than others. The tree that has been built incrementally may not be the most efficient for the following reasons:

1. **Poor Structure.** As discussed in Section 6.7, the shape of the hyperregions in a *k*d-tree can affect the nearest neighbour search. These can only be chosen well if the tree is built with knowledge of all the data which it is to contain: this is clearly not possible if points are added incrementally, when the *k*d-tree's structure will be determined entirely according to the order in which the points arrived.

2. **Imbalance.** A robotic system is likely to spend periods of time in very similar areas of the state space. This can mean very uniform data arriving which causes serious imbalance.

3. **Dead Points.** Points are marked as dead when they are old and inaccurate. From then on they remain in the tree, wasting memory and occasionally increasing search time. This is particularly serious after a sudden change which results in large regions of the SAB-tree being killed.

The garbage collection operation happens when the robotic system is not in use. It flattens the SAB-tree into a list of exemplars, removes the dead points and then rebuilds the SAB-tree using the balance-inducing and hyperrectangle-choosing algorithm of Section 6.7. The theoretical cost is

$$O((k_s + k_a + k_b)(N_D + N \log N)) \tag{7.18}$$

where N is the number of live points and N_D the number of dead points. There is an amusing analogy between this garbage collection and human sleep; one purpose of the latter is believed to be the removal of extraneous links between memories or knowledge acquired during the day.

Chapter 8

SAB Control

A computationally cheap method for learning world models has been proposed and developed in Chapters 5—7. This chapter shows how to use the world models to make control decisions using a probabilistic heuristic. It then analyses and illustrates the method.

8.1 Making a Control Decision

It is now time to review the control cycle first introduced in Section 4.4. This cycle controlled a dynamic task while maintaining and improving its *SAB* world model. Since then, Chapter 5 explained how and why the nearest neighbour generalization is to be used to represent the model. Chapters 6 and 7 showed to to update and access this world model, in the possible presence of disorder. The emphasis was on performing these operations efficiently. This chapter addresses the question of how the world model should be used. It will consider a general method to implement Step 3 of the *SAB* control cycle, repeated here for convenience in Table 8.1.

To motivate this section, I provide an illustration from the simple Mountain Car domain of Section 4.3. In this example, the state is the distance from the start, the action is the distance that the pedal is pressed and the behaviour is the perceived horizontal velocity. Imagine that learning has just begun, and the task is to guide the mountain car from state $s = 0.5$km to state $s = 9.5$km at a constant velocity $b = 5.6$ units. An error of ± 0.2 units is considered acceptable. So far, there have been four *SAB* learning cycles which have produced these four exemplars:

$$\mathbf{E} = \{(0.5, 4.8 \rightarrow 5.7), (1.8, 6.0 \rightarrow 6.3), (2.7, 7.5 \rightarrow 5.1), (1.3, 9.0 \rightarrow 3.5)\} \qquad (8.1)$$

The situation is depicted in Figure 8.1, with the exemplars positioned in **State** $\times$ **Action** space. The polygons separate the various nearest neighbour regions. The vertical line denotes the current state $\mathbf{s}_{\text{current}} = 3.4$.

1.	Observe current perceived state $s_{current}$
2.	Receive task specific goal behaviour b_{goal} from higher level of control. The requested behaviour might depend partially on $s_{current}$.
3.	**Access the current world model to obtain a raw action a_{raw} which is predicted to be likely to achieve b_{goal} acting in the current state.**
4.	Apply action a_{raw}.
5.	Observe actual behaviour b_{actual}.
6.	Update the world model with the information that $(s_{current}, a_{raw}) \rightarrow b_{actual}$.

Table 8.1: The *SAB* Control Cycle

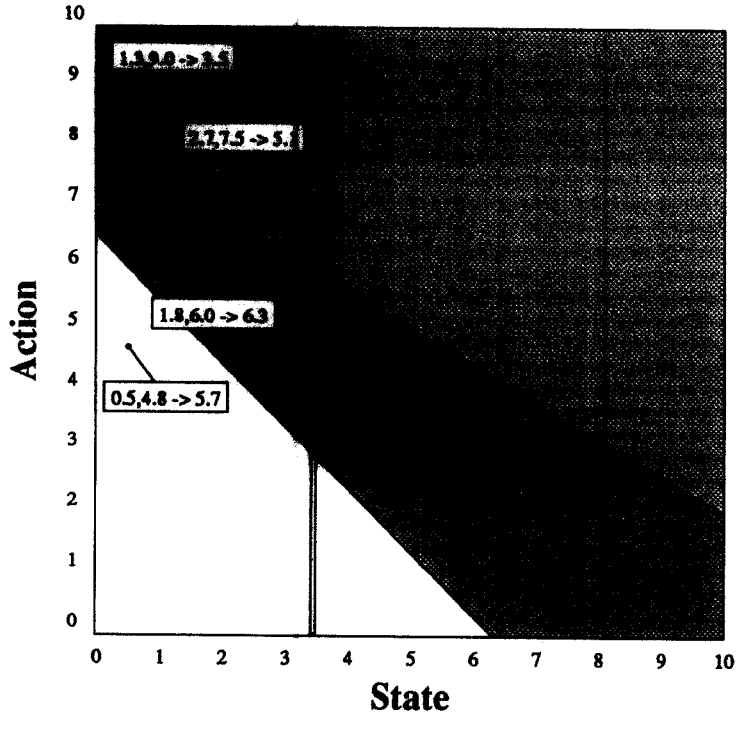

Figure 8.1

State × Action space with the regions denoting the behaviour predicted by the nearest neighbour. The brighter the region, the closer it is to the goal behaviour of 5.6. Now we want to find x to give $(3.4, x \rightarrow 5.6)$.

At this point the controller must decide what to do. If it chooses a low-valued action then the state-action pair it applies will be in the lower bright region, which is predicted to be very good, with behaviour 5.7. But in many ways, an action near the most recent action (which produced behaviour 5.1) has a prediction which looks more *reliable* because it is closer. As a result, despite the predicted behaviour being inferior, it might be preferable (we will see later that this is what partial inversion would recommend). But then, perhaps it would be a mistake to try the same action again: if the controller never tries anything new, how can it hope to improve?

8.1.1 The Utility of Information

There are two opposing aims in making a control choice. These are

1. **Perform.** We wish to perform as well as possible given the knowledge contained in the performance element.

2. **Experiment.** In order to perform better in future, it is worth trying actions with no guaranteed success, but with the chance of a valuable discovery.

Aim 1 represents doing the best for the system immediately and Aim 2 represents an investment in the future. This is the same decision as that of a corporate director deciding whether to pay the shareholders a dividend, or provide extra finances for the company's R & D division. The system must take into account the *utility of information.*

Given a SAB-tree representation of the PSTF, the **perform** mode of operation can be easily achieved by partial inversion. There might be some experience in the SAB-tree in which an action applied in a state close to the current state had produced behaviour close to that of the goal. If this is so, then partial inversion will obtain this action. If such previous experience is not available, partial inversion may suggest something sensible, but for reasons described in Section 5.1 is not reliable. Figure 8.2 shows the action recommended by partial inversion for the exemplar set of Equation 8.1.

It is immediately clear that it is not desirable for the learning controller to either perform only Aim 1 or Aim 2. If it only ever tried what it knew best there would be no diversity of experience and learning would not occur. If it only ever experimented there would be a large body of knowledge, but no improvement in performance.

8.1.2 Random Experimentation

One approach to this dilemma is to have two modes of operation. The first is experimental, where actions are chosen randomly with no reference to previous experience. The second is demonstration-mode in which no chances are taken, and the best known action is always used. These modes are altered by an intelligent supervisor which can judge when further experimentation is needed and when it is positively unwelcome. An effective example of these two modes of operation is Mel's MURPHY [Mel, 1989]. This learns the inverse

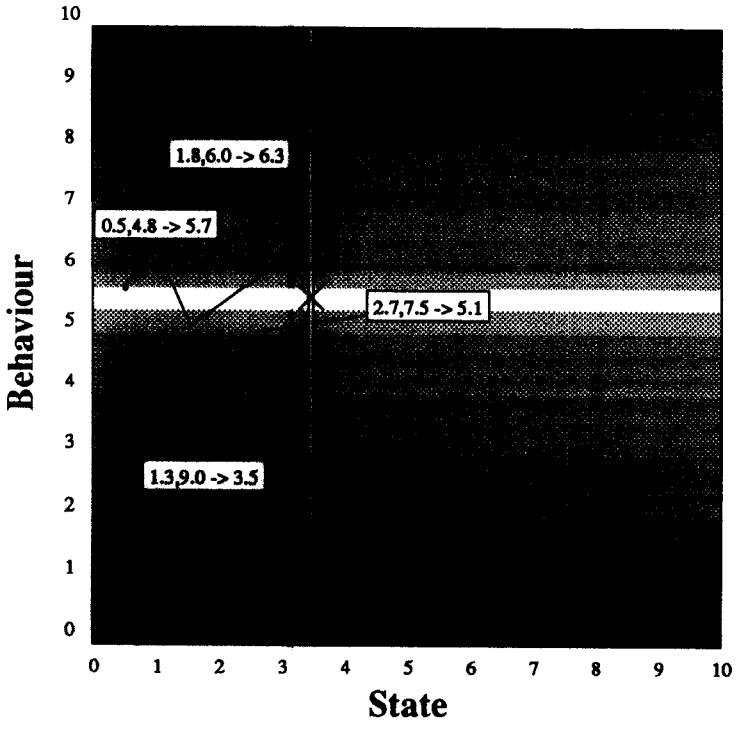

Figure 8.2

Exemplars in **State** × **Behaviour** space. The regions denote the action recommended by partial inversion. The shading denotes the desirability of various behaviours. Partial inversion to obtain behaviour 5.6 finds the nearest neighbour to the cross, and thus recommends action 7.5.

differential kinematics by firstly "flailing" the arm in front of its camera and then making plans based on the results of the observations.

Instead of requiring an intelligent supervisor to decide when it has finished learning, the controller can occasionally test itself.

Let us call this method 'R'-control (Random decisions). The mountain car learning task was run with alternate trials of perform and experiment. On every odd numbered trial, the controller simply chose random actions. On every even numbered trial, the controller, on each occasion, chose the best known action. The choice was by means of partial inversion.

The learning curve for this scheme is shown in Figure 8.3. The learning run was was repeated forty times. Each learning run started with no world knowledge, and then performed eighty trials. Each trial was an attempt at the mountain car task. The car was started at state 0.5km and actions were executed until the state was greater than 9.5km. An action was successful if it resulted in a speed between 5.4 units and 5.8 units. The correct trajectory is obtained by the set of actions which keep the state in the white zone of the diagram. On each odd trial, random actions were taken, and on each even trial the best known actions were taken. For each even trial, the number of unsuccessful action choices was recorded. For each odd, random, trial the results were discarded. For the nth trial, the graph shows the mean number of errors for the forty occasions of trial n. The vertical bars display sample standard deviations (for explanation of the confidence-level key, see Appendix A).

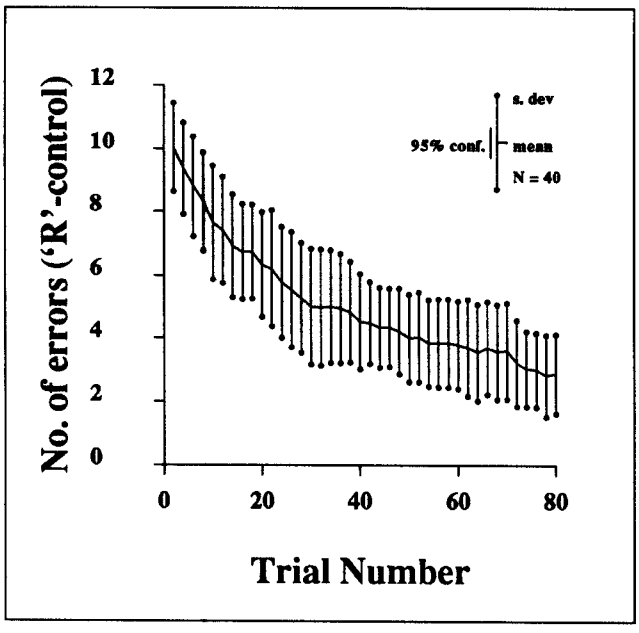

Figure 8.3

Performance against trial number for a controller which, on alternate trials, performs and well as possible and experiments randomly. This data is obtained from forty independent learning runs.

This graph indicates that the 'R'-controller does indeed learn, but that even after eighty trials the performance tends to be poor. The learning rate is slow because, for each area of the state space, the controller must wait until the random experiments happen to obtain a correct action. Some experiments at some states might be lucky early, others very unlucky, perhaps for all eighty trials. The slow learning is directly related to the large amount of useless data collected. Figure 8.4 shows the distribution of exemplars in the *SAB*-tree after one such run of eighty trials, and it can be seen to be almost uniform. As I explained in Chapter 5, a uniform distribution of exemplars can lead to a very slow rate of learning. Recalling from Chapter 5 how badly the nearest neighbour generalization performs in uniform distributions, it is perhaps surprising that learning occurs at all. The reason it does occur is that the mountain car uses only a two-dimensional control space. It seems unlikely, in a control space of higher dimension, that the method would learn to a tolerance as accurate as this within an attainable number of trials.

8.1.3 Local Random Experimentation

An improvement is the 'L'-controller. This makes local random perturbations of the best known action instead of choosing entirely random actions. The first action, in the first trial, is chosen entirely at random, because with no experience there can be no best known action. Figure 8.5 shows the performance, with identical experimental conditions to those of the 'R'-controller. Its learning is superior, and occasionally it achieves perfect behaviour within eighty trials. There is still great deviation in learning speeds. The speed depends

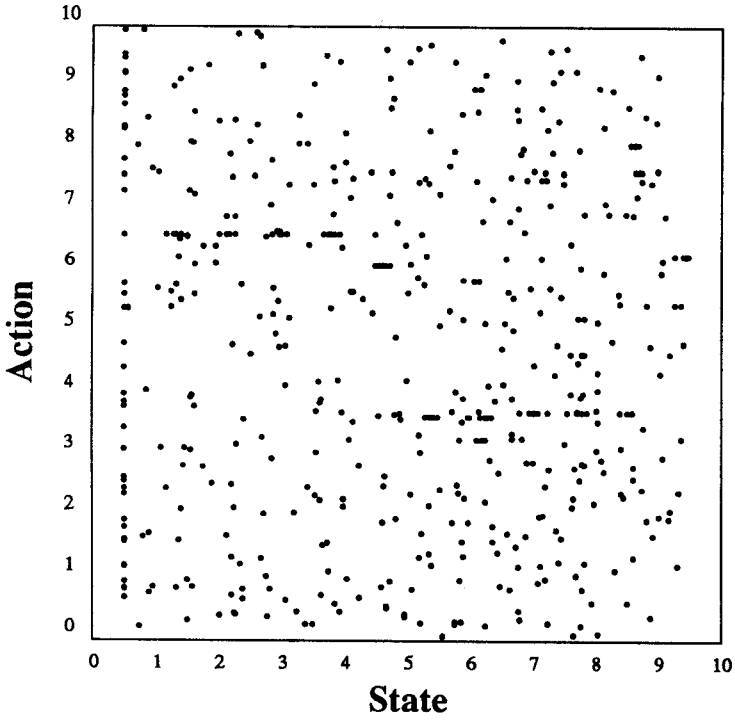

Figure 8.4

The distribution of exemplars in **State** × **Action** space after learning using the 'R'-controller.

on the very first randomly chosen action. If it happens to be very wrong, then it will take many trials until it is adjusted enough to be right. If it is lucky with the first random guess, then learning will proceed much more quickly.

There is also another danger with 'L'-control which was escaped here: that it might fall into a local minimum at some area of the state space. This would occur if the best known action is better than its surrounding actions, without being globally best. In a slightly more complex domain, such local minima are possible. A controller which has a danger of never escaping local minima is not robust.

A combination of 'R'-control and 'L'-control could be designed which, when experimenting, would occasionally choose locally and occasionally globally. This is similar to the solution which simulated annealing uses to avoid local minima in its hill climbing. The learning behaviour is likely to lie between that of 'R' and 'L'-control, with the danger of local minima removed.

8.1.4 Sceptical Experimentation

The control schemes mentioned up to now have the problem that learning does not occur *during* a trial. During odd-numbered trials there are occasions when the 'R' or 'L'-controller should know what should be done next but wastefully ignores it, instead trying some different random action. Conversely, on some even trials there are occasions when the controller should know that the action suggested by partial inversion is not good enough. On such occasions, to repeat the same action is to repeat a known error.

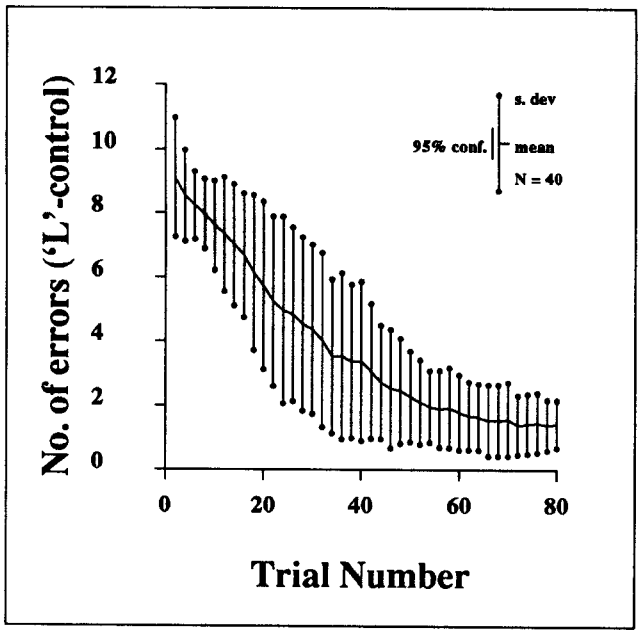

Figure 8.5

Performance against trial number for a controller which, on alternate trials, performs as well as possible and experiments with local perturbations. This data is obtained from forty independent learning runs.

Instead, because the exemplar set can be used for prediction as well as for partial inversion, a candidate action can be *checked* before its application. Thus the use of two modes of operation can be dispensed with, and instead the best known action can, on all occasions be considered, but only as a *candidate* action. Such a sceptical controller only then experiments when the candidate action is predicted to be unsuccessful. Figures 8.6 and 8.7 shows the results of two sceptical controllers. 'SR'-control uses the best known action when it predicts success, and chooses a random experiment otherwise. 'SL' is similar except that it perturbs the best action when experimenting.

These methods perform better than 'R'-control and 'L'-control respectively. 'SL'-control achieved perfect performance within eighty trials on almost half of the learning runs. This success motivates an attempt to try to squeeze further information out of the exemplar set.

8.1.5 SAB Action Chooser

The new scheme will again be content if the predicted behaviour of the candidate action is successful. However, if not it will choose an alternative action with more care. Instead of choosing the first random action which it generates, it generates several random actions, and in turn uses its world model to predict the result of using them. How should it rank different candidate actions?

One idea is to choose the action with the best predicted behaviour. Upon inspection, this loses its appeal. If nothing has worked well in the controller's experience, then despite

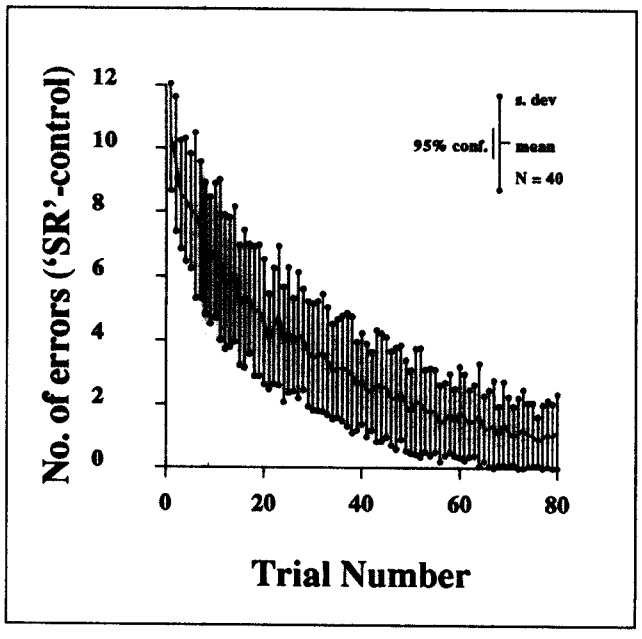

Figure 8.6

Performance against trial number for a sceptical controller ('SR'-control). This checks the best recommended action before use, and if success is not predicted, tries something random.

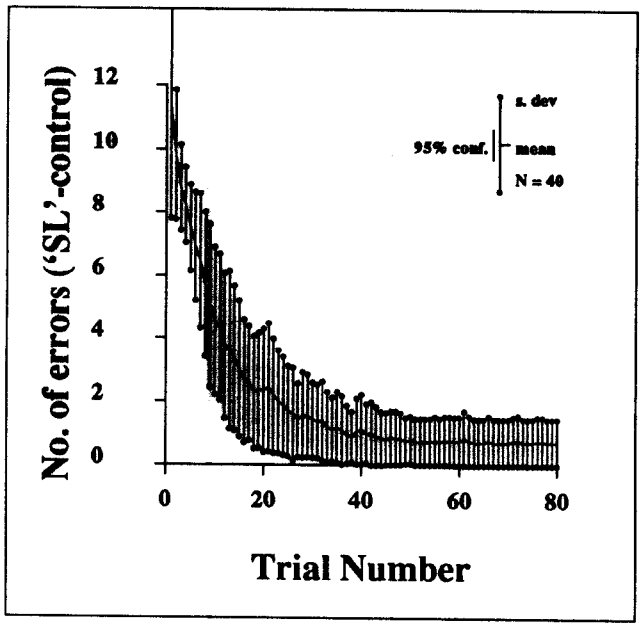

Figure 8.7

Performance against trial number for a sceptical controller ('SL'-control). This checks the best recommended action before use, and if success is not predicted, perturbs the action.

its inadequacy, the action with the best predicted behaviour is likely to be close to or at the initial candidate action obtained from partial inversion. Thus the controller will consistently choose something which it knows will perform mundanely in preference to actions which it, less reliably, predicts to be worse.

Fortunately, the *reliability* of predictions can also be estimated thanks to the explicit exemplar set representation. The closer the nearest neighbour which has been used to make the prediction, the more reliable it is. A desirable qualitative ranking of candidate actions is:

1. (Best) The candidate action is predicted to be successful, and is reliable.
2. The candidate action is predicted to be successful, but the prediction is unreliable.
3. The candidate action is not predicted to be successful, but this prediction is not reliable.
4. (Worst) The candidate action is reliably predicted to be unsuccessful.

I will now introduce a quantitative heuristic which achieves this qualitative ranking. First, it should be made clear what constitutes successful behaviour. The *SAB* cycle definition states that the higher level controller specifies a target behaviour $\mathbf{b}_{\text{goal}}$. If success is ever to be attainable, some small deviation from this goal must be deemed acceptable. Depending on the task, different accuracies might be adequate. Furthermore, different components of the goal behaviour may be needed to different accuracy. To specify this, the controller also supplies a *Dim*(**Behaviour**)-dimensional tolerance vector, $\boldsymbol{\tau}$. It is not necessary for the tolerance to be varied during learning—if high accuracy is required a high tolerance can be specified right from the start. Behaviour $\mathbf{b}$ *achieves behaviour* $\mathbf{b}_{\text{goal}}$ *to tolerance* $\boldsymbol{\tau}$ iff

$$\forall i \quad [\mathbf{b}_{\text{goal}}]_i - \tau_i < \mathbf{b}_i < [\mathbf{b}_{\text{goal}}]_i + \tau_i \tag{8.2}$$

where $\mathbf{b}_i$ is the ith component of vector $\mathbf{b}$.

The ranking heuristic can be achieved quantitatively by some elementary decision analysis. Actions are ranked by their estimated *probability of success.*

To make this estimate, we have at our disposal the nearest neighbour in $\mathbf{E}$ to $(\mathbf{s}_{\text{current}}, \mathbf{a})$. Call this exemplar $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}^{\text{smooth}}_{\text{near}})$. We model the unknown behaviour at the point $(\mathbf{s}_{\text{current}}, \mathbf{a})$ as a random variable influenced by this nearest neighbour. Call this random variable B. It varies over the space of real-valued *Dim*(**Behaviour**)-dimensional vectors. The expected behaviour is $\mathbf{b}^{\text{smooth}}_{\text{near}}$. The reliability of the expected behaviour is high if the exemplar is close, and it is low if the exemplar is distant. This is modelled by setting the standard deviation of the variable to be proportional to the distance to $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}})$. Call the constant of proportionality C. There is no other useful information which can be obtained from the nearest neighbour, and so to fully specify the random variable, this heuristic will assume it is distributed normally with independent vector

components. We are now in a position to quantify the probability of success. Let us first consider the chance that the ith component is successful. This probability is denoted by

$$P^i_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \tau_i, \mathbf{E}) \tag{8.3}$$

which we will define as the probability that applying action $\mathbf{a}$ in the current state $\mathbf{s}_{\text{current}}$ will cause the ith component of the resulting behaviour to be successful. Writing the ith component of the resulting behaviour as B_i, then "ith component success" means that B_i is within tolerance, or symbolically:

$$[\mathbf{b}_{\text{goal}}]_i - \tau_i < B_i < [\mathbf{b}_{\text{goal}}]_i + \tau_i \tag{8.4}$$

The probability of success will be computed solely in terms of the exemplar set $\mathbf{E}$. To summarize, we have defined

$$\begin{aligned} &P^i_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \tau_i, \mathbf{E}) \\ &= \mathbf{Prob}([\mathbf{b}_{\text{goal}}]_i - \tau_i < B_i < [\mathbf{b}_{\text{goal}}]_i + \tau_i \mid \mathbf{s}_{\text{current}}, a, \mathbf{E}) \end{aligned} \tag{8.5}$$

The probability is computed with respect to the nearest neighbour of $(\mathbf{s}_{\text{current}}, \mathbf{a})$. Call this neighbour $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}^{\text{smooth}}_{\text{near}})$. Then, as described above, we assume the random variable B_i is distributed with expectation $\left[\mathbf{b}^{\text{smooth}}_{\text{near}}\right]_i$ and standard deviation $\sigma = C \mid (\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}) - (\mathbf{s}_{\text{current}}, \mathbf{a}) \mid$. Thus the probability that $[\mathbf{b}_{\text{goal}}]_i - \tau_i < B_i < [\mathbf{b}_{\text{goal}}]_i + \tau_i$ is the area under the appropriately transformed normal distribution curve in Figure 8.8, and can be computed as

$$\begin{aligned} &P^i_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \tau_i, \mathbf{E}) \\ &= \text{erf}\left(\frac{[\mathbf{b}_{\text{goal}}]_i + \tau_i - \left[\mathbf{b}^{\text{smooth}}_{\text{near}}\right]_i}{\sigma}\right) - \text{erf}\left(\frac{[\mathbf{b}_{\text{goal}}]_i - \tau_i - \left[\mathbf{b}^{\text{smooth}}_{\text{near}}\right]_i}{\sigma}\right) \end{aligned} \tag{8.6}$$

Where $\sigma = C \mid (\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}) - (\mathbf{s}_{\text{current}}, \mathbf{a}) \mid$, and $(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}^{\text{smooth}}_{\text{near}})$ is the nearest neighbour to $(\mathbf{s}_{\text{current}}, \mathbf{a})$ in the state and action components. The erf function is defined by

$$\text{erf}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2} dt \tag{8.7}$$

We have computed the chance that the ith component is successful, P^i_{succ}. Now, under the independence, assumption we can calculate the actual probability of success:

$$P_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \tau, \mathbf{E}) = \prod_{i=1}^{Dim(B)} P^i_{\text{succ}}(\mathbf{a}, \mathbf{s}_{\text{current}}, [\mathbf{b}_{\text{goal}}]_i, \tau_i, \mathbf{E}) \tag{8.8}$$

To illustrate the use of the heuristic, imagine that **State**, **Action** and **Behaviour** are all 1-d spaces, and that we are in state $\mathbf{s}_{\text{current}} = 3.3$ and the desired $\mathbf{b}_{\text{goal}}$ is 11. Assume

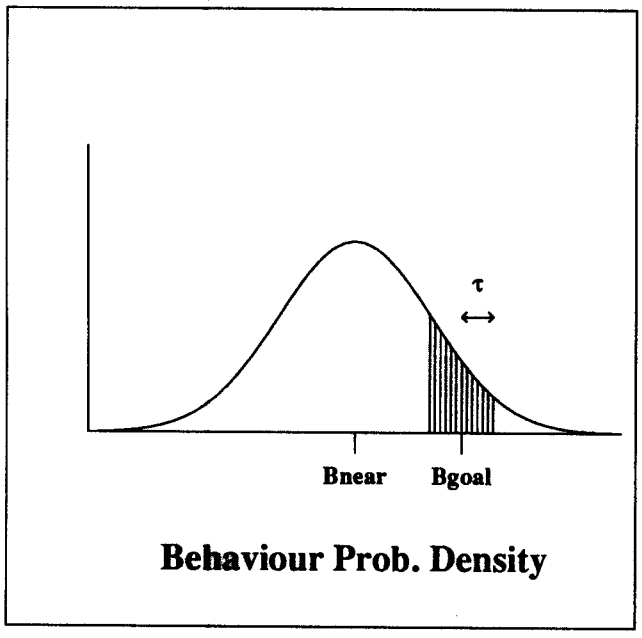

Figure 8.8

The P_{succ} estimate. The probability of success is the area of the shaded region.

further that the only previous experience is that when $\mathbf{s}_1 = 3$ and $\mathbf{a}_1 = 7$ then $\mathbf{b}_1 = 10$. The use of the heuristic, with a tolerance of 0.5, is demonstrated in Figure 8.9. It compares three possible actions and chooses one which is fairly near to 7, so that the behaviour will be fairly near to 10 (and thus might be close to the goal, 11), but not so near to 7 that the behaviour will surely be extremely close to 10.

We are now also able to inspect how the heuristic behaves for the control choice problem posed earlier in this section by Figure 8.1. The graph in Figure 8.10 shows how the probability of success varies with different candidate actions. The constant C is again chosen as 1. It shows that an action very close to the partial-inversion-recommended action is favoured ($\mathbf{a}_{raw} = 7.5$). This is reasonable, because the *state* is fairly different from any state yet experienced, and so by repeating the same action, there is not a loss of diversity of experience. To show what would happen if the state *were* similar, I move the current state shown in the original example of Figure 8.1 back so it is very close to the exemplar ($2.7, 7.5 \rightarrow 5.1$). I make the current state 2.9. The graph of the new probability of success, in Figure 8.11, indicates that the candidate action of 7.5 is now regarded very poorly. This is because it is so close to an earlier experience which did not achieve the required behaviour.

This new, probabilistic, control choice algorithm is summarized in Table 8.2. In future it will be called the *SAB* action chooser. Before performing empirical tests, let us consider some of its expected benefits.

- As we shall see later, the heuristic is not sensitive to the choice of C (the propor-

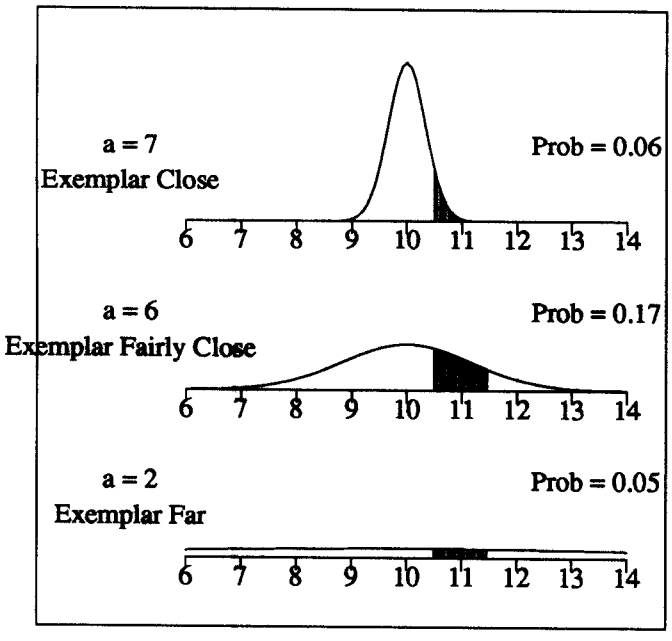

Figure 8.9

Choosing an action which is likely to cause behaviour in the range 10.5 to 11.5. Top: Action very close to earlier exemplar in which behaviour was 10. Middle: Action fairly close. Bottom: Action very different.

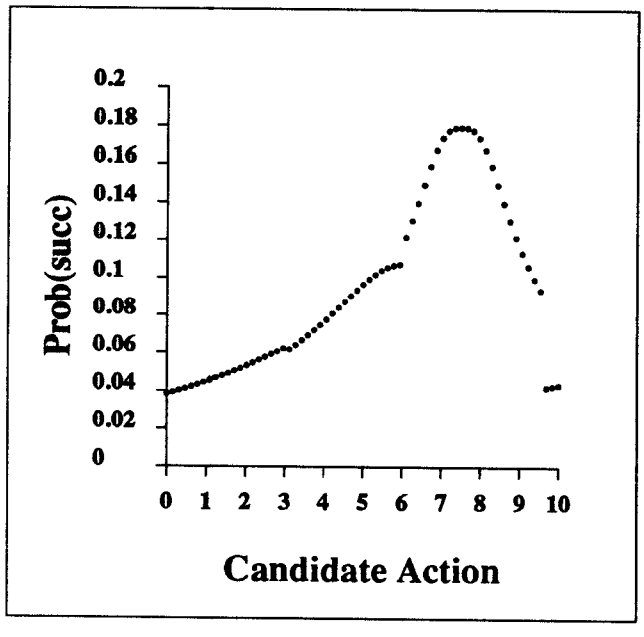

Figure 8.10

The probability of success heuristic judging the candidate actions for the situation depicted in Figure 8.1. The heuristic has discontinuities between nearest neighbour regions. For example, the state-action pair $(3.4, 9.6)$ has a different nearest neighbour than $(3.4, 9.2)$, resulting in a very different estimate.

Algorithm:	*SAB* action chooser
Input:	$\mathbf{s}_{\text{current}}$, a *Dim*(**State**)-dimensional vector of real numbers $\mathbf{b}_{\text{goal}}$, a *Dim*(**Behaviour**)-dimensional vector of real numbers **E**, of type **exemplar-set** τ, a *Dim*(**Behaviour**)-dimensional vector of real numbers **number-cands**, of type **integer**
Output:	$\mathbf{a}_{\text{raw}}$, a *Dim*(**Action**)-dimensional vector of real numbers **probsucc**, of type **real**
Pre:	$\mathbf{E} \neq \phi$
Post:	**probsucc** $= P_{\text{succ}}(\mathbf{a}_{\text{raw}}, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \tau, \mathbf{E})$ There is also the informal requirement that the probability of success be relatively high.
Code:	
1.	$(\mathbf{s}_{\text{pi}}, \mathbf{a}_{\text{raw}}, \mathbf{b}_{\text{pi}}, [\mathbf{b}_{\text{pi}}]_{\text{smooth}})$:= a nearest neighbour to $(\mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}})$ in (state,behaviour) space
2.	$(\mathbf{s}_{\text{near}}, \mathbf{a}_{\text{near}}, \mathbf{b}_{\text{near}}, \mathbf{b}_{\text{near}}^{\text{smooth}})$:= a nearest neighbour to $(\mathbf{s}_{\text{current}}, \mathbf{a}_{\text{raw}})$ in (state,action) space
3.	**probsucc** $:= P_{\text{succ}}(\mathbf{a}_{\text{raw}}, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \tau, \mathbf{E})$
4.	If $\mathbf{b}_{\text{near}}^{\text{smooth}}$ is successful with tolerance τ then return $\mathbf{a}_{\text{raw}}$ and **probsucc**
5.	for **i** := 1 until **number-cands** do
5.1	$\mathbf{a}_i$:= randomly generated action
5.2	$\mathbf{p}_i := P_{\text{succ}}(\mathbf{a}_i, \mathbf{s}_{\text{current}}, \mathbf{b}_{\text{goal}}, \tau, \mathbf{E})$
5.3	If $\mathbf{p}_i >$ **probsucc** then
5.3.1	**probsucc** := $\mathbf{p}_i$
5.3.2	$\mathbf{a}_{\text{raw}} := \mathbf{a}_i$
6.	return $\mathbf{a}_{\text{raw}}$ and **probsucc**

Table 8.2: The *SAB* action chooser algorithm

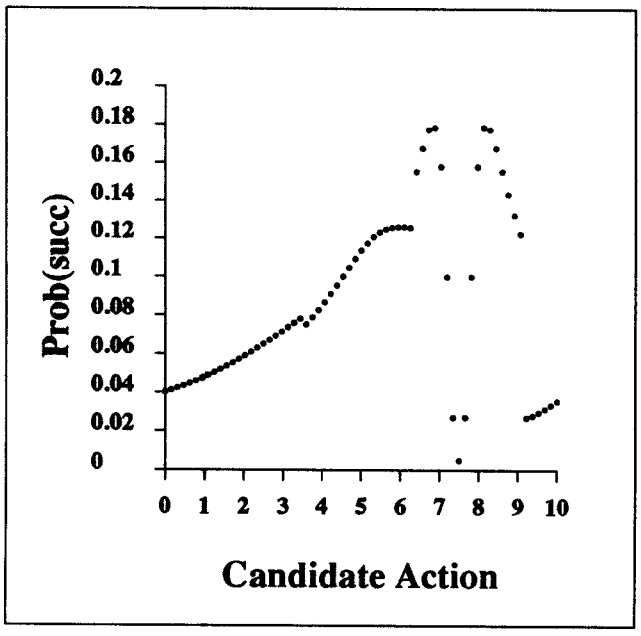

Figure 8.11

The probability of success heuristic judging the candidate actions for an altered version of the situation depicted in Figure 8.1. The alteration is that the current state is 2.9 instead of 3.4.

tionality constant between nearest neighbour distance and standard deviation). The ranking between good and bad candidate actions is changed very little as C varies.

- Initially it is likely that for many states the only previous experience will be negative. Then this heuristic favours those actions which are as far away as possible from any yet applied. This is because the probability of success for actions which are close to a previous unsatisfactory behaviour is extremely low, while actions far away have a probability of success which is merely low. Thus a wide variety of actions will be generated until some relatively promising ones are discovered.

- If the promising actions were misleading, for example in a local minimum, then upon repeated trials, the heuristic would eventually once again favour further points. This is because the vicinity will gather many exemplars, and so any candidate action in the vicinity will be close to a promising, but unsuccessful, action. Because it is close, the reliability of the prediction will outweigh the favourable closeness to the goal behaviour.

- The partial inversion action can also be included among the candidate actions. If it is ranked as best, despite not having been predicted as successful, this merely indicates that the pessimistic prediction was unreliable. There is no danger of stuck states: if wrong, then next time we will know reliably that it is unsuccessful, meaning a very low estimated probability of success.

- The candidate actions need not be generated from a uniform distribution, provided that all actions have a finite probability of being generated. A non-uniform distribution, biased in favour of the actions close to the partial-inversion-recommended action, may in some circumstances be advantageous. This is analysed in the Section 8.2.

- If the task is in any way repetitive then the system will eventually return to the same neighbourhood of state space, and require the same goal behaviour. Let us assume that on the original occasion the action chooser had been successful. On the new occasion none of this computation is needed. Partial inversion immediately provides the action, which is then validated by a nearest neighbour prediction. So, for a repetitive task, the computational burden decreases as the behaviour improves.

- If there is known noise in the function, then even with the assumption that the $\mathbf{b}_{\mathbf{near}}^{\mathbf{smooth}}$ component is an accurate value of the expected behaviour, there is still some variation to be anticipated if the same state-action pair were to be repeated. This could be included in the P_{succ} heuristic by defining the standard deviation as

$$K + C \mid (\mathbf{s}_{\mathrm{near}}, \mathbf{a}_{\mathrm{near}}) - (\mathbf{s}_{\mathrm{current}}, \mathbf{a}) \mid \tag{8.9}$$

  where K is the estimated standard deviation of noise. In this investigation this possibility has not been implemented because an explicit record of local variance is not stored.

The empirical trial of the mountain car task using the *SAB* action chooser, 'SAB'-control, was performed in the manner of the earlier trials. The results are given in Figure 8.12. On each cycle, the number of candidate actions was five. They were chosen randomly from a uniform distribution over the space of actions.

In all forty learning runs, this controller achieved perfect behaviour before the 80th trial. This is evidence of the power which can be obtained from the nearest neighbour generalization. Once again, there is initially wide variation in the performance, between trials 1–20. Figure 8.13 shows the distribution of exemplars after learning with 'SAB'-control. They are clustered around the solution. It is expected that this effect would be more pronounced and more important in a higher dimensional control space.

I have defined a measure for scoring candidate actions, but have not provided a method for obtaining that which has the highest score. This would require a search of all possible actions, which for a realistic space of possible actions (we are assuming a continuum of vectors), real time response does not permit. Instead, the favourite of a sample of randomly generated actions is used. A large sample can be expected to provide an action close to that which would be recommended by exhaustive search, but with the penalty of more computation per action. An important question is how does performance vary as the number of candidates used varies, and also as their distribution varies. This is difficult to

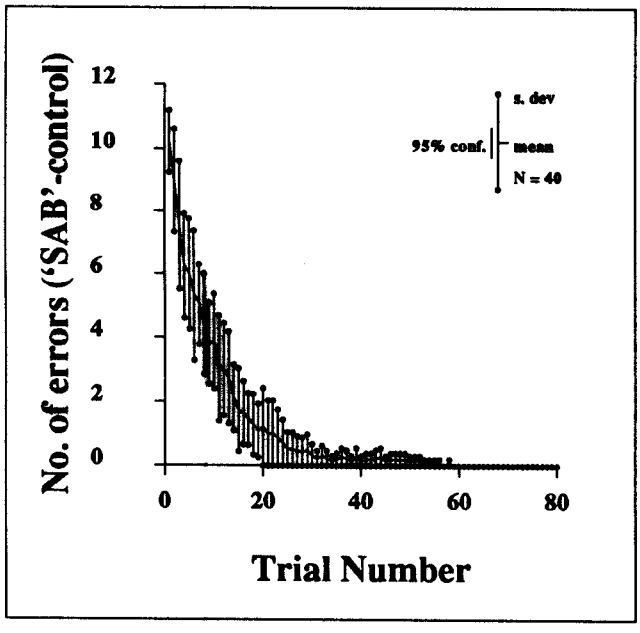

Figure 8.12

The performance against trial number for a sceptical controller, which when confronted with a predicted failure uses the best out of five randomly generated candidate actions.

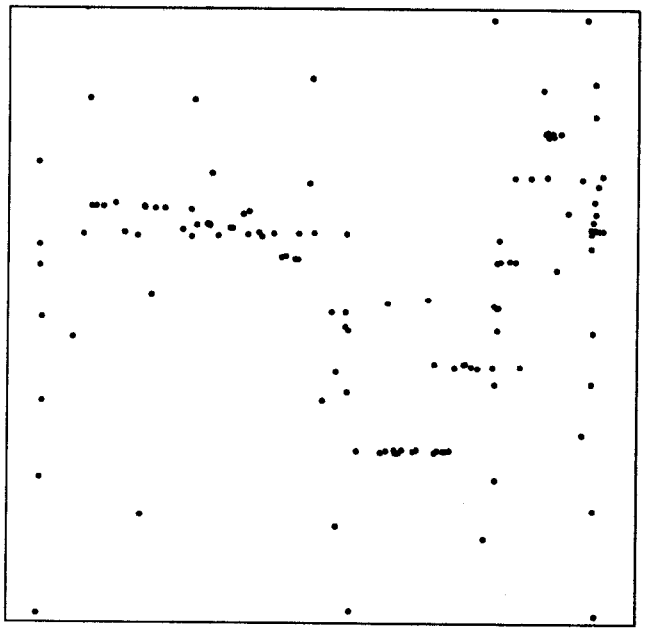

Figure 8.13

The distribution of exemplars in **State** × **Action** space after learning using the 'SAB'-controller.

analyse formally and so will be discussed later on the basis of empirical observations from the results chapter. Here we will be content to make a few informal observations.

Firstly, consider the case where the action is only ever chosen from a small number of candidates. Provided at least one candidate action is generated from a distribution which has non-zero density over the entire action space, then if one performs a series of trials all with the same $\mathbf{s}_{\text{current}}$ and $\mathbf{b}_{\text{goal}}$, then (subject to $\mathbf{b}_{\text{goal}}$ being attainable) $\mathbf{a}_{\text{raw}}$ will eventually converge to a value which achieves the tolerance. Informally the proof, which requires general continuity, would be similar to that in Section 5.5, which assumes exemplars are drawn uniformly and randomly. Instead, the exemplar generation is biased so that, in the absence of success, they are positioned far apart from one another. This will not increase the time until the whole action space is covered with exemplars to distance $\epsilon = \mid \tau \mid /G$, and so the time needed is, if anything less than that for a uniform distribution.

Secondly, in the limit we would not expect the learning performance to increase greatly with a large increase in the number of candidates. Consider using the best of a million candidate actions. This is likely to be very close to the best possible action to maximize the heuristic. But this heuristic is itself only an approximation, and a small change in, for example, the C parameter, or the assumed normal distribution, would change the best recommended action by more than one part in a million. The use of the heuristic is to choose actions significantly better than randomly. It is not expected to prescribe success.

At first it seems too good to be true that there exists an optimization procedure which is guaranteed to avoid local minima. The reason is that unlike most optimization procedures, this is using unbounded storage, and is thus modelling the whole function, rather than merely keeping a record of the local search state.

The previous few paragraphs have considered the case of repetitive occasions in which the same state and required behaviour occur. When we apply an action and discover it was unsuccessful we cannot immediately find out what would have happened had we tried something else—the state, $\mathbf{s}_{\text{current}}$, of a dynamic manipulator cannot be expected to remain constant. The failure or success of the control choice of the previous time step may no longer be relevant. It is a result of the explicit storage of the exemplars that the performance can nevertheless be expected to improve, because information from all those occasions in the learning history which *are* currently relevant will still be available.

8.2 Control Decision Analysis

In this section I perform some elementary analysis to show, subject to some constraints on the continuity and slope of the PSTF, that the *SAB* action chooser method optimizes the chosen action very quickly. I will make some simplifying assumptions, then model the optimizing behaviour. I will then discuss the relationship between the analysis of the simplified system and the expected performance of the real action chooser.

The simplifications are

1. Assume the set of actions and the set of behaviours are both one-dimensional.

2. Assume that on each cycle we have the same state, and require the same behaviour.

3. Assume we only have a small memory: we recall only the most promising exemplar to date.

4. Assume we manage to choose a candidate action with the highest heuristic probability of success.

It will become apparent that given the pessimistic assumption 3, the optimistic assumption 4 is easy to satisfy.

The *SAB* action chooser is now posed as an optimization method, and we can analyse how quickly this optimization converges. The exemplar set **E** contains only the exemplar $(s_{\text{current}}, a_i, b_i, b_i^{\text{smooth}})$ which has had the most successful behaviour to date. We can dispense with the b_i^{smooth} component because, if there is only one exemplar, $b_i^{\text{smooth}} = b_i$. Given a candidate action we can use the probability of success estimate of Section 8.1.

$$P_{\text{succ}}(a, s_{\text{current}}, b_{\text{goal}}, \tau, \{(s_i, a_i, b_i)\})$$

$$= \quad \text{erf}\left(\frac{b_i + \tau - b_{\text{goal}}}{Cx}\right) - \text{erf}\left(\frac{b_i - \tau - b_{\text{goal}}}{Cx}\right) \tag{8.10}$$

$$\text{where} \quad x = \mid a - a_i \mid$$

In this equation, x is the distance of the candidate action from the best action yet seen. If b_i had been successful, then the maximum probability of success occurs with this distance set to zero and thus with a repetition of the previous action a_i. More rigorously, as $x \to 0$, then $P_{\text{succ}} \to \text{erf}(\infty) - \text{erf}(-\infty) = 1 - 0$.

If b_i is not successful then the probability of success of repeating a_i is zero. For example, if $b_i > b_{\text{goal}} + \tau$, then $P_{\text{succ}} \to \text{erf}(\infty) - \text{erf}(\infty)$ as $x \to 0$. Thus

$$x = 0 \Rightarrow P_{\text{succ}} = 0 \tag{8.11}$$

For a distance greater than zero, the probability of success estimate will be positive, but as the distance gets very large, then the probability of success will again tend to zero because as $x \to \infty$, $P_{\text{succ}} \to \text{erf}(0) - \text{erf}(0)$. Thus

$$x = \infty \Rightarrow P_{\text{succ}} = 0 \tag{8.12}$$

The P_{succ} estimate is a smooth function of x, and so there must be an intermediate optimum distance. Writing $e_i = b_i - b_{\text{goal}}$ as the behaviour error of the current exemplar,

then the optimum occurs when

$$\begin{aligned}
&\frac{\partial P_{\text{succ}}}{\partial x} = 0 \\
\Rightarrow\ &\frac{\partial}{\partial x}\left[\text{erf}\left(\frac{e_i+\tau}{Cx}\right) - \text{erf}\left(\frac{e_i-\tau}{Cx}\right)\right] = 0 \\
\Rightarrow\ &-\left(\frac{e_i+\tau}{Cx^2}\right)\exp\left[-\left(\frac{e_i+\tau}{Cx}\right)^2\right] + \left(\frac{e_i-\tau}{Cx^2}\right)\exp\left[-\left(\frac{e_i-\tau}{Cx}\right)^2\right] = 0 \\
\Rightarrow\ &\ln\left(\frac{e_i+\tau}{e_i-\tau}\right) = \frac{4e_i\tau}{C^2x^2} \\
\Rightarrow\ &x = \frac{2}{C}\sqrt{\frac{e_i\tau}{\ln\left(\frac{e_i+\tau}{e_i-\tau}\right)}}
\end{aligned} \tag{8.13}$$

So the ideal distance, or *step size*, at which to try the next action can be computed. This recommended step size is a function of the current error in observed behaviour, and of the tolerance. If the error is large with respect to the tolerance, then

$$\ln\left(\frac{e_i+\tau}{e_i-\tau}\right) = \ln\left(1+\frac{2\tau}{e_i-\tau}\right) \approx \frac{2\tau}{e_i-\tau} \approx \frac{2\tau}{e_i} \tag{8.14}$$

Thus, using Equation 8.13, the ideal distance is

$$x \approx \frac{2}{C}\sqrt{\frac{{e_i}^2\tau}{2\tau}} = \frac{\sqrt{2}\mid e_i \mid}{C} \tag{8.15}$$

Thus, if the current error is large compared with the tolerance, the ideal optimization step is proportional to the current error. At each cycle there will be two possible actions to try next, one distance x to the left of a_i and one distance x to the right. The actual action tried is selected at random. If there is improvement then the current best action is updated, else it remains the same. If either $a_i - x$ or $a_i + x$ leads to improved behaviour then within an expected two time steps, improved behaviour will be obtained.

If neither $a_i - x$ or $a_i + x$ give an improvement then either (i) the algorithm has used too large a step size or (ii) the algorithm is in a local minimum. This simplified *SAB* action chooser would then never progress. We will discuss later the behaviour of the real algorithm in this circumstance.

If, however, progress is made, how quick is it? It would be useful to derive the expected increase in accuracy per step. Unfortunately, to obtain this, a further assumption is needed, that the PSTF has locally a strictly positive (or strictly negative) slope, bounded below by $G_{\text{min}} > 0$ and above by $G_{\text{max}} > 0$.

$$\forall a_1 > a_2 \quad (a_1 - a_2)G_{\text{min}} < b_1 - b_2 < (a_1 - a_2)G_{\text{max}} \tag{8.16}$$

where $b_1 = \text{PSTF}(a_1)$ and $b_2 = \text{PSTF}(a_2)$. Assume without loss of generality that b_i, the behaviour of the most promising exemplar, is less than b_{goal}. Then, unless one of

the above problems occur, the addition of the computed step length to a_i will result in a chosen action that gives improved behaviour. Using the approximation of Equation 8.15, we have

$$a_{i+1} = a_i + \frac{\sqrt{2}}{C} \mid e_i \mid \tag{8.17}$$

But by the assumption of bounded slope, we can deduce that

$$b_i + (a_{i+1} - a_i)G_{\min} < b_{i+1} < b_i + (a_{i+1} - a_i)G_{\max} \tag{8.18}$$

where $b_{i+1} = \text{PSTF}(a_{i+1})$. Therefore, from Equation 8.17,

$$b_i + G_{\min}\frac{\sqrt{2}}{C} \mid e_i \mid < b_{i+1} < b_i + G_{\max}\frac{\sqrt{2}}{C} \mid e_i \mid \tag{8.19}$$

We are interested in $e_{i+1} = b_{i+1} - b_{\text{goal}}$, the error of the next time step. It is not known whether it will be positive or negative, but in either case its magnitude is bounded above. In the first case, where $b_{i+1} \leq b_{\text{goal}}$,

$$\mid e_{i+1} \mid = b_{\text{goal}} - b_{i+1} < b_{\text{goal}} - b_i - G_{\min}\frac{\sqrt{2}}{C}(b_{\text{goal}} - b_i) = (1 - G_{\min}\frac{\sqrt{2}}{C}) \mid e_i \mid \tag{8.20}$$

In the second case, where $b_{i+1} > b_{\text{goal}}$,

$$\mid e_{i+1} \mid = b_{i+1} - b_{\text{goal}} < b_i - b_{\text{goal}} + G_{\max}\frac{\sqrt{2}}{C}(b_{\text{goal}} - b_i) = (G_{\max}\frac{\sqrt{2}}{C} - 1) \mid e_i \mid \tag{8.21}$$

In either case $\mid e_{i+1} \mid < \mid Ke_i \mid$ where $K = \max\{G_{\max}\frac{\sqrt{2}}{C} - 1, 1 - G_{\min}\frac{\sqrt{2}}{C}\}$. The error $\mid e_{i+1} \mid$ is a constant factor less than $\mid e_i \mid$, provided that both $(G_{\max}\frac{\sqrt{2}}{C} - 1) < 1$ and $(1 - G_{\min}\frac{\sqrt{2}}{C}) < 1$. This implies that C must be greater than $G_{\max}/\sqrt{2}$. This, in turn agrees with the informal statement of the previous section that the value of C should reflect the expected slope of the function. If C is suitable then the error decreases exponentially with the number of steps of optimization.

In summary, if the function is well behaved in the region of interest, then if the initial error is large with respect to the tolerance, then each time progress is made

$$\mid e_{i+1} \mid < K \mid e_i \mid \tag{8.22}$$

where K is defined above in terms of the function's slope and the system parameter C. If the choice whether to increase or decrease the current best known action is made randomly then progress can be expected once every two time steps. Thus after a number of time steps logarithmic in e_0/τ (the ratio between the initial error and the required error) the behaviour is expected to be successful.

Now let us consider what would happen to the algorithm if it *does* remember and reuse all its experience. It will be shown that the optimization can still achieve an exponential decrease in the error, even when the forgetful method would have stopped improving.

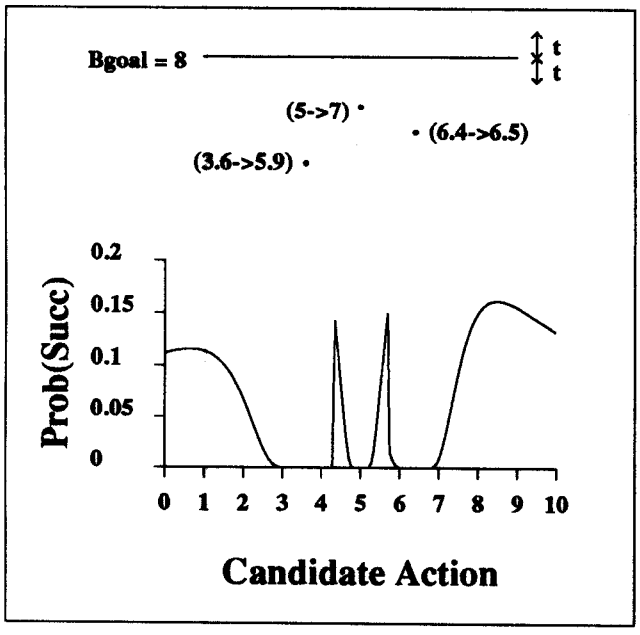

Figure 8.14

Three state-free exemplars have been experienced, none achieving within tolerance 0.5 of the target behaviour of 8. The action component of each exemplar is aligned with the action axis of the graph. There are now four local maxima in the P_{succ} estimate.

Improvement would have stopped if the two actions $a_{f1} = a_i - \frac{\sqrt{2}}{C}e_i$ and $a_{f2} = a_i + \frac{\sqrt{2}}{C}e_i$ had both produced behaviour inferior to that of a_i. Let us consider the case where three exemplars are available for use by the P_{succ} estimate. An illustration is given in in Figure 8.14, in which the initial action $(5 \rightarrow 7)$ was not successful, but the two exemplars at the computed ideal distance were also unsuccessful, and both worse. After this situation, the probability of success for all other candidate actions is plotted in the same figure. As is exemplified here, there are precisely four local maxima in the P_{succ} estimate. These occur at the following points:

$$a_{c1} = a_{f1} - \frac{\sqrt{2}}{C}\sqrt{\frac{(b_{f1} - b_{\text{goal}})\tau}{\ln\left(\frac{b_{f1} - b_{\text{goal}} + \tau}{b_{f1} - b_{\text{goal}} - \tau}\right)}}$$

$$a_{c2} = (a_{f1} + a_i)/2$$

$$a_{c3} = (a_{f2} + a_i)/2$$

$$a_{c4} = a_{f2} + \frac{\sqrt{2}}{C}\sqrt{\frac{(b_{f2} - b_{\text{goal}})\tau}{\ln\left(\frac{b_{f2} - b_{\text{goal}} + \tau}{b_{f2} - b_{\text{goal}} - \tau}\right)}}$$

The best of these four actions will be attempted. Candidates 2 or 3 are chosen if the P_{succ} estimate suspects that there is some successful behaviour between a_i and either a_{f1} or a_{f2} respectively. Thus, if the step size had been too big, the step size would be reduced to half its previous value. If the resulting behaviour looked promising, search would continue, either as in the original linear search, or if step lengths were continually overestimated, then as a binary chop. In either case the error is reduced by a constant multiplicative factor at each step.

Candidates 1 or 4 are chosen if the behaviour at a_i is so unpromising that is is preferable to be a large distance from the inferior a_{f1} or a_{f2} to remaining in the presence of a_i. This can be viewed as an escape from a local minima.

Thus, for one dimension, the behaviour of this optimization technique is robust: it does not get stuck in local minima, but when the function is well behaved, the number of bits of accuracy increases linearly. The actual error thus decreases exponentially. Unfortunately, this has assumed a dimensionality of only 1 and has used the simplification that the chosen action is always the most promising according to the P_{succ} estimate. I shall now give informal explanations as to why the desirable behaviour can be expected to scale up.

- **Many Dimensions.** The action space is not as large as the control space, but it can be expected to be up to six-dimensional for a fully orientable robot arm. With more than one dimension there is a continuum of candidate actions at the ideal distance, instead of merely the two in the previous analysis. However, if the function is well behaved in the sense of Equation 8.16 then half the candidate actions can be expected to improve behaviour and half to degrade it. The expected number of actions needed to improve behaviour is still only two. For a high dimensional space of behaviours the expected decrease in error is, however, less than for one dimension.

- **Maximizing the P_{succ} estimate is too expensive.** This analysis has assumed that of those points which maximize the P_{succ} estimate, one was chosen as the next action. In practice, this can only be approximated. The approximation consists of the best of a number of randomly chosen actions. The distribution is biased in favour of actions which have the computed ideal distance. Thus it can be hoped that the best action approximates one of the actions with a relatively high probability of success and so in turn it can be hoped that the local search may still be linear in the number of bits of accuracy—this hope will be tested in the results chapter.

The reasoning of this section has lead to the following heuristic for choosing candidate actions. Some are generated locally, and some globally. The global actions are generated from a uniform distribution over the whole action space. Equation 8.15 indicated that a good place to look for candidates was within a distance proportional to the current error. There are two system parameters for the random candidate generator: P_{loc}, the probability of choosing a candidate action from the local distribution, and C_{loc}, the constant of proportionality used when an action *is* generated locally.

8.3 Learning One's Own Strength

The *SAB* action chooser does not get stuck in local minima, and thus, on repeated occasions, will eventually search the entire action space to any granularity until the behaviour is found. In some respects this is an advantage, but there is also a significant drawback. If the behaviour is not attainable then an enormous variety of actions will be tried unnecessarily. Each time one of these hopeless attempts is made, it is not guaranteed that the behaviour which is produced is as close to the goal as it can get. This is precisely because the *SAB* action chooser is avoiding local minima.

This section does not offer a general solution to this difficult problem. Instead it is suggested that for many tasks one of the two following proposals can be used.

- **Extra domain knowledge.** The designer can use their own knowledge of the system to ensure that all behaviours which will be requested are indeed attainable.
- **Heuristic estimate of attainability.** The controller can estimate from the world model whether it believes the proposed behaviour is possible.

One primitive mechanism for achieving the second solution is considered in Appendix C. If the behaviours have a *direction* then given a requested behaviour direction and a behaviour magnitude, the heuristic computes an estimated probability that the magnitude can be achieved. The higher level controller can use the heuristic to decide whether to request the behaviour.

This is related to a difficult problem of statistical decision theory: the "two armed bandit", reviewed in [Kaelbling, 1990a]. If one is told one may have a large number of goes on a two armed bandit in which each arm pays out with different probability, what should be done to maximize the expected payoff? Continue pulling the arm which first pays out? Perform three hundred trials of each and then always pull the one which performs best? The current consideration is similar. At what point should the *SAB*-control choice mechanism resign itself to never being able to supply the requested behaviour, and instead always supply the best known? If it resigns too early then it might miss trying an action which would have actually achieved the behaviour. Thus learning will have become "stuck" at an error. If it resigns too late then during the search the experimental actions are likely to degrade task performance far more severely than if the best known action were used.

Kaelbling, in [Kaelbling, 1990b], deals with this problem for the extreme case in which the controller has no world model at all, and is instead trying to maximize a *payoff* signal. Her work suggests some algorithms which are effective for worlds specified by binary-valued input vectors, and which empirically tend to find high payoffs. Although the learning algorithms are computationally expensive and the assumption is of a small finite space of actions, it would certainly be interesting to apply similar techniques to this problem.

Chapter 9

Learning to Perform a Task

This chapter discusses a scheme by which SAB world models and the SAB action chooser can be integrated into a variety of tasks. A naive control method called "ice puck control" is introduced. It is then explained (i) how low level tasks can be controlled directly by the SAB action chooser and (ii) how some medium level tasks can be indirectly controlled using "ice puck control" and then (iii) how a large compound task can be modularized into a hierarchy of learning subtasks.

9.1 Where do Goal Behaviours come from?

The previous chapters have introduced a robust and efficient method to quickly learn the dynamics of a possibly changing world. They have also shown how to use this model: when a goal behaviour is required, a raw action signal is obtained which will either achieve the behaviour or else gain useful information.

Let us reconsider Step 2 in the original learning controller scheme, shown again in Table 9.1. There is no general way to achieve this step because there is no generic task. In this chapter I will propose that for many of the common tasks which we might expect a robot to perform, the generation of goal behaviours is easy.

The specification of robotic tasks is similar to the specification of software systems. It occurs, at least initially, in an abstract and rather informal form, perhaps as a piece of text, or simply an idea. The design of a method to execute the task is again similar to the design of a program: it is broken up into smaller subcomponents, with correspondingly less abstraction. Different programs often share some features, such as graphical data display, and similarly many robot tasks have common subcomponents. Some of these were mentioned in Chapter 2. They include components such as sensing, trajectory following, and holding still. At the lower levels, the design of software and robot task design differ: at an intermediate level of abstraction, the human programmer can write down the program, and then the *compiler* converts these intermediate notations to the entirely concrete form of executable machine code. The designer of a robotic system has to go down to the

1.	Observe current perceived state $s_{current}$
2.	**Receive task specific goal behaviour b_{goal} from higher level of control. The requested behaviour might depend partially on $s_{current}$.**
3.	Access the current world model to obtain a raw action a_{raw} which is predicted to be likely to achieve b_{goal} acting in the current state.
4.	Apply action a_{raw}.
5.	Observe actual behaviour b_{actual}.
6.	Update the world model with the information that $(s_{current}, a_{raw}) \rightarrow b_{actual}$.

Table 9.1: The *SAB* Control Cycle

concrete level manually. This work has been investigating the possibility of automating the very bottom level, world modelling, but I wish to show here that it can make higher levels easier as well. To motivate this, I will describe how to connect the *SAB* control cycle to various tasks. I will demonstrate that many aspects of an abstract task definition can remain abstract.

The chapter will begin with a diversion—the control of an extremely elementary dynamic system.

9.2 Controlling an Ice Puck is Easy

The argument that is it easy to control an ice puck makes a substantial jump in the train of thought of the dissertation. The reason for the effort expended here will become clear in the next section, which will coalesce the ideas of *SAB* control choice, and ice puck control.

9.2.1 An Idealized Ice puck

Figure 9.1 shows an ice puck: a small one kilogram mass sitting on a frictionless surface. It has two forces acting on it, F_1 and F_2, F_1 shown acting horizontally and F_2 vertically on the diagram. Because it is idealized, its equations of motion are very simple:

$$\begin{aligned} a_1(t) &= F_1(t)/M \\ a_2(t) &= F_2(t)/M \end{aligned} \tag{9.1}$$

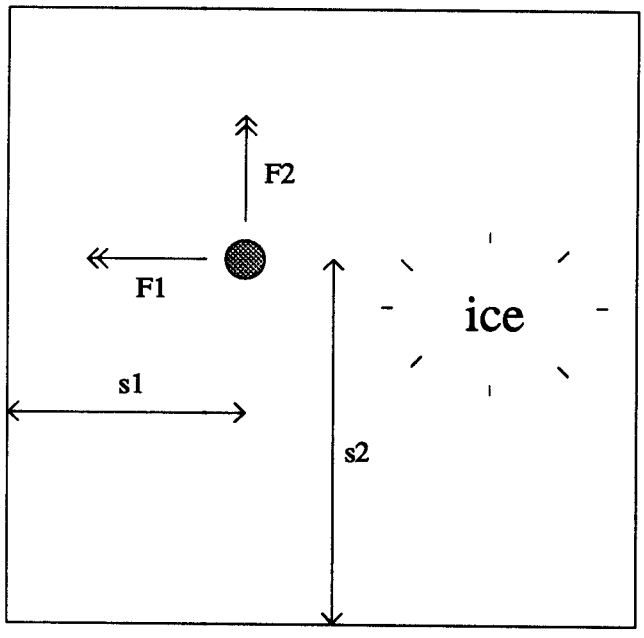

Figure 9.1

The ice puck world. The puck moves across an idealized frictionless surface, controlled by two orthogonal force vectors.

where $M = 1$, a_1 is the horizontal acceleration and a_2 is the vertical acceleration. Accelerations and forces are thus equivalent, so ice puck control is merely the application of accelerations. Let us write the state of the ice puck as $\mathbf{s}(t) = (s_1(t), v_1(t), s_2(t), v_2(t))$, where s_i is the distance from the origin in the ith direction, and v_i is the ith component of velocity.

From the definitions of position, velocity and acceleration, it is simple to write down the behaviour of the puck at any future time as a function of all the future accelerations. We can do this for each component independently.

$$\begin{aligned} v_i(T) &= v_i(0) + \int_0^T a_i(t)dt \\ s_i(T) &= s_i(0) + \int_0^T v_i(t)dt \end{aligned} \tag{9.2}$$

The ice puck is controlled at discrete time steps of length h. During each time step a constant acceleration is supplied. Write $s_i[n]$ as the position at the start of the nth time step, $v_i[n]$ as the velocity, and $a_i[n]$ as the acceleration applied during the nth time step. Then

$$\begin{aligned} v_i[n] &= v_i[n-1] + ha_i[n-1] \\ &= v_i[0] + h\sum_{j=0}^{n-1} a_i[j] \end{aligned} \tag{9.3}$$

and

$$\begin{aligned} s_i[n] &= s_i[n-1] + \int_0^h (v_i[n-1] + a_i[n-1]\,t)\,dt \\ &= s_i[n-1] + hv_i[n-1] + \tfrac{1}{2}h^2 a_i[n-1] \\ &= s_i[0] + h\sum_{j=0}^{n-1}\left(v_i[j] + \frac{1}{2}ha_i[j]\right) \\ &= s_i[0] + h\sum_{j=0}^{n-1}\left(v_i[0] + h\sum_{k=0}^{j-1} a_i[k] + \frac{1}{2}ha_i[j]\right) \\ &= s_i[0] + nhv_i[0] + h^2\sum_{j=0}^{n-1}(n-j-\frac{1}{2})a_i[j] \end{aligned} \tag{9.4}$$

After n steps of constant acceleration α, the state of the ith variable is

$$\begin{aligned} v_i[n] &= v_i[0] + nh\alpha \\ s_i[n] &= s_i[0] + nhv_i[0] + \tfrac{1}{2}\alpha n^2 h^2 \end{aligned} \tag{9.5}$$

It is thus not difficult to predict future behaviour of the idealized ice puck. It is also elementary algebra to control the ice puck in order to make it reach a goal position and goal velocity at a goal time step, t_g. Call this combination of objectives a ***puck goal.*** All that is necessary is, for each direction of the puck world, to invent a trajectory which will achieve the appropriate component of the puck goal.

One such simple trajectory is bang-bang control, shown in Figure 9.2. It consists of applying a constant acceleration α for N time steps and then acceleration $-\alpha$ for $t_g - N$ time steps. There are two values to choose, and two constraints to meet: the values of $s_i[t_g]$ and $v_i[t_g]$. Thus the values can be derived by elementary algebra. Interested readers may inspect the results at the end of this section.

Instead of generating the trajectory and then tracking it, the controller computes the initial recommended acceleration, applies it, and then discards it. On the next time step it implements the same procedure. This is possible, because the trajectory is cheap to compute: one square root and approximately ten multiplications are required per puck component. In this manner the trajectory can be automatically adjusted to take account of earlier tracking errors.

There are many other ways of choosing accelerations. For example, one could find accelerations α and β, such that applying α for time $\frac{1}{2}t_g$ followed by β for time $\frac{1}{2}t_g$ achieves the desired state. The bang-bang method has two particularly desirable properties.

- The maximum magnitude of any acceleration on the trajectory is as low as possible. For example, imagine that there was a constraint that no component of the acceleration applied to the puck could have magnitude greater than $A_{\max}$. If the controller

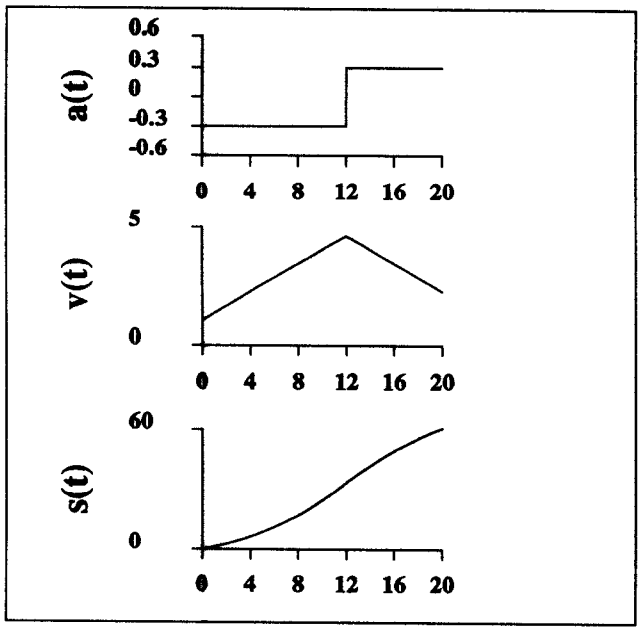

Figure 9.2

The puck starts in state $s = 0, v = 1$ and wishes to achieve state $s = 60.8, v = 2.2$ in twenty time steps of one second. The solution is acceleration $\alpha = 0.3$ for 12 time steps and acceleration $-\alpha = -0.3$ for eight steps.

receives a puck goal, and the bang-bang computation produces an acceleration which is too large, then no other trajectory would have managed either.

- The controller dynamically recomputes the trajectory at each time step, and this computation is affordable. If the computed acceleration is applied accurately then the next dynamically computed trajectory will match the first. This property does not generally hold for other kinds of trajectory.

There is also a potential disadvantage. The sudden switch in requested accelerations would, in a non-ideal world, be hard to achieve and difficult to control. In practice, many robot trajectory generators are designed to produce paths which in terms of position are smooth to five levels of differentiation.

The conclusion of this section is that given a goal state and time, it is easy to achieve the goal for an idealized ice puck. In Sections 9.3 and 9.4 I explain why the *SAB* model of the world permits other dynamic systems to be controlled in the same manner.

9.2.2 Generating the Bang-Bang Trajectory

All control variables are treated independently. Let the current position and velocity be (s_0, v_0), and the goal be (s_g, v_g) after t_g time steps. This analysis finds the values N, α and β such that when acceleration α is performed for N time steps and acceleration β for $N - t_g$ time steps, the goal state will be obtained. These values will be chosen to make α and β close in magnitude while ensuring that N is integral.

First we find X, possibly non-integral, and γ, such that accelerating with γ for Xh seconds and then accelerating with $-\gamma$ for $(t_g - X)h$ seconds will achieve the goal. Such values would satisfy the two equations

$$v_g = v_0 + h\gamma(2X - t_g) \tag{9.6}$$

$$s_g = s_0 + hv_0t_g + \frac{1}{2}h^2\gamma(-t_g^2 + 4Xt_g - 2X^2) \tag{9.7}$$

Thus, γ in terms of X is, from Equation 9.6, $\gamma = (v_g - v_0)/(h(2X - t_g))$. Substituting into Equation 9.7 gives

$$(2X - t_g)(s_g - s_0 - hv_0t_g) = \frac{1}{2}h(v_g - v_0)(-t_g^2 + 4t_gX - 2X^2) \tag{9.8}$$

This quadratic equation is solved by

$$X = t_g - \frac{R}{h} \pm \frac{1}{2h}\sqrt{4R^2 - 4Rht_g + 2h^2{t_g}^2} \tag{9.9}$$

where

$$R = \frac{s_g - s_0 - hv_0t_g}{v_g - v_0} \tag{9.10}$$

N is then chosen as the closest integer to X, and then the accelerations α and β are computed to solve the goal position, time and velocity. The solution for α, which is then used as the recommended acceleration, is:

$$\alpha = \frac{2(s_g - s_0)}{h^2Nt_g} - \frac{v_0(t_g + N) + v_g(t_g - N)}{hNt_g} \tag{9.11}$$

9.3 Low Abstraction Tasks

A "low abstraction task" is defined here to be a task which has been specified to sufficient detail that the controller is able to compute the goal behaviour directly from the task description. This level is the bottom level that the system designer must consider—this is a considerable improvement on the conventional bottom level of abstraction.

9.3.1 Achieving a Fixed Behaviour

The Mountain Car domain provided an example of this kind of task in Chapter 8, where the goal was to continue moving at a constant velocity. The domain was constructed so that the velocity was the behaviour being modelled. There is no difficulty integrating the *SAB* action chooser into such a task. At each control cycle, the requested behaviour remains fixed. This continues until some higher level of control, perhaps a human operator, is satisfied. The success of the task can be judged as either the sum of the deviations from the goal behaviour, or the number of cycles which were in error.

9.3.2 Following a Perceived Trajectory

For a dynamic system, in which the Perceived State Transition Function is being learned, the behaviour vectors are perceived accelerations. The task can be controlled as if accelerations were being applied directly to the perceived state. In fact they will be applied indirectly, by requesting the *SAB* action chooser to try to achieve them.

The trajectory task can thus pretend that it is an ice puck controller. Put more formally, it can imagine it is an entirely linear and decoupled system. On each cycle, and for each perceived position component, it considers the current position and velocity. It also considers the desired position and velocity specified by the trajectory for the next time step. Can this one-time-step puck goal be achieved? If there is a solution to the one-step puck goal, then the component of the goal acceleration has been obtained. It might, however, not be possible because the current velocity might not allow positions to be matched on the next cycle if velocities are also to be matched. In that case the puck goal of two steps in the future is used. The required acceleration is then handed to the *SAB* action chooser.

If the current position and velocity were to mismatch badly with those desired then the requested accelerations would be unattainably large. The action chooser would on repetitions of the same situation try a wide variety of actions. This is because all previous occurrences would have failed, and thus the highest probability of success would be achieved by trying something as different as possible. The large variety of low probability actions might degrade performance, and could not be expected to bring the state any closer to the trajectory. In this case it might be preferable that the *SAB* action chooser always produced accelerations as close as it could to those requested. Empirically, this problem was not observed in the experiments detailed in Chapter 10. However, if it were likely to be a problem it could be avoided by use of domain knowledge to specify limits on accelerations. A request would then never be made to the *SAB* action chooser for any larger acceleration. Alternatively, the maximum accelerations could be estimated by a heuristic such as that described in Appendix C.

9.4 Middle Abstraction Tasks

Some middle abstraction tasks can be immediately transformed into low abstraction tasks. These include the "hold still" task, the "move here" task and, at a slightly more abstract level, the "move here quickly" task.

9.4.1 Hold Still

This can be implemented as a request to follow the stationary trajectory. The position at which to remain can be specified as the position when the "hold still" task was first activated.

9.4.2 Move Here

In the "move here" task, the position, velocity and time to achieve the goal are specified. This is then an ice puck goal of Section 9.2. The correct accelerations can be derived accordingly.

9.4.3 Move Here Quickly

Commonly, one does not wish to specify the time the robot should take to get to a particular configuration, but instead would prefer it to arrive reasonably quickly. This could be done by simply guessing a time or having a default time for movements, but this could lead to inefficiencies, either because the robot will have moved unnecessarily slowly, or because it tried to achieve too fast a trajectory and overshot the target, because of inadequate deceleration. A naive controller which consistently made this mistake could oscillate unstably if it persisted in trying to get to the goal at a speed which exceeded the capabilities of the actuators.

Instead, the controller uses its knowledge about its strength. The algorithm performs a binary chop to find a suitable movement time. When a movement time is considered, the ice puck controller is consulted to determine the accelerations which would be necessary to achieve the movement time. Then the *SAB* world model is consulted to determine the probability that the necessary accelerations can be attained. The shortest time with a sufficiently high probability of success is used. This decision could be formalized if it were provided with (i) the expected cost of trying a trajectory which is not attainable and (ii) the expected payoff from using short times.

Experiments with such a controller are documented in Section 10.4. The advantage of the autonomous time choice controller is that it has some abstractness: the detail of how long the movement should take is computed automatically. Its performance on repeated trials can be expected to find a trajectory which is not unacceptably slow. However, it would certainly be unlikely to find the theoretically optimal trajectory. If such a trajectory were required then a more computationally expensive controller would be necessary.

9.5 Compound Tasks

Many interesting tasks have several components. In this section I discuss, by means of an example, how *SAB* learning is integrated with multiple-component tasks. The example is volleying a ball. The physical situation is depicted in Figure 9.3.

The ball on the right of the diagram is fired towards the two-jointed arm on the left of the diagram. The arm must hit the ball so that it lands in the bucket. The arm and ball are both acting under gravity. The controller can sense the following data (in perceived coordinates):

- The position and velocity of the hand.

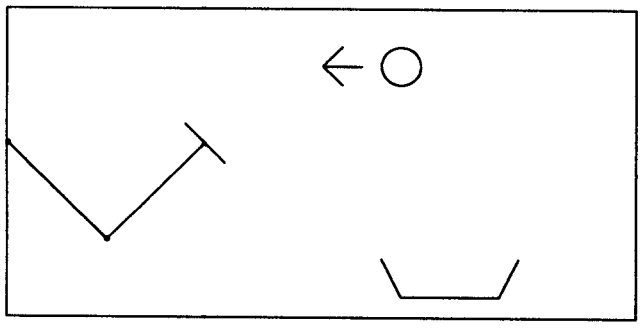

Figure 9.3

The volley task: physical situation.

- The position and velocity of the ball on the following occasions: (i) when it is fired, (ii) when it is hit and (iii) when it lands.
- The position of the bucket.

The control consists of the torques sent to the shoulder and elbow joint of the arm. The bat is fixed at right-angles to the forearm. The details of the simulation parameters are given in the results chapter.

One approach to using *SAB* learning to control this task would be to specify it as a nine-dimensional state space. The state vector has four components for the arm state, four components for the ball state and one component for the position of the bucket. The action is the two-component vector of the joint torques supplied, and the behaviour is the change in arm state and in ball state.

The PSTF of such a system would have the strange property that four components of the state, namely those of the ball state, are almost never affected by the action. If the *SAB*-controller were given the goal system-state of "anywhere" for the arm and "at rest, y = 0, x = bucket position" for the ball, then generally there would be absolutely no action which could achieve it and very occasionally all actions would achieve it. Thus local control, specified by local goal behaviours, to send the ball to the bucket would be entirely useless. Instead, there would have to be some global level to the controller, much as was used for ice puck goals, to derive the final ball position. Such a global level could, for example, search a large number of time steps ahead to find if any of the actions currently available could, with adequate subsequent actions, achieve the final goal. A more computationally tractable alternative would be a dynamic programming approach. In principle, if the PSTF is learned to sufficient accuracy, then the controller will succeed at the task. A large amount of data would be needed because the controller would not know in advance which parts of the control space would be important. In the absence of other domain knowledge, no data would be useful until the goal had been achieved at least once—for the volley problem this would almost certainly take a very long time.

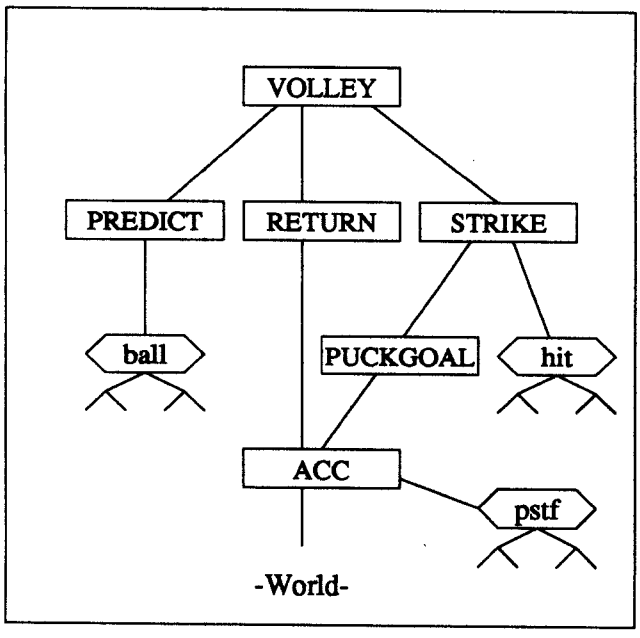

Figure 9.4

The volley task decomposed into a small hierarchy of subtasks.

The problem with the formulation as a nine-dimensional state space system was that the structure inherent in the problem was flattened away. It is clear that a human given the volley task would make some abstract, symbolic organization of the task into components. In particular they would notice that the final behaviour of the ball depends on whether it contacts the bat, and that if it does contact the bat, the bat's behaviour at the time will affect the ball's subsequent behaviour.

If a robot is to achieve the task, it would be an interesting exercise to let it try to deduce this relationship from the observed data. Some recent work by [Sammut and Michie, 1989] investigates discovering such qualitative descriptions of dynamic systems from raw behavioural data. However, for a large class of cases, such as this volley example, domain knowledge such as this is easy for a human designer to express. It is similarly easy to give an abstract, qualitative description of how to perform the entire task.

> Predict where the ball will be when it comes into convenient hitting range. Make sure the hand is in the same position at the same time, and moving at such a speed as to cause the ball to land in the bucket. After the ball and bat have collided, move the arm to a waiting position.

This abstract volley task can be separated into three slightly less abstract subtasks, indicated in Figure 9.4. `Predict` estimates the ball's behaviour. `Strike` brings the bat to contact the ball at a controlled position and speed. `Return` holds the bat steady after the strike.

The `Predict` task models the perceived behaviour of the ball prior to being hit. This model is learned by a *SAB*-tree, which estimates the time until the ball arrives within range of the arm and the state of the ball at this point. The ball is defined to be in range of the arm when it reaches a distance of 1.5m from the shoulder (the stretched arm is 2m long). The *SAB*-tree is a mapping

$$\underbrace{\text{Ballstart}}_{S} \times \underbrace{()}_{A} \rightarrow \underbrace{\text{Ballhit} \times \text{Timehit}}_{B}$$

which is updated once every trial. There is no control over this aspect of the ball's behaviour so **Action**, the space of control actions, is empty.

The `Return` task simply computes a perceived acceleration to thrust the endpoint towards the stationary waiting position. This is an instantiation of the "hold still" task described in the previous section. The torques to achieve this acceleration are computed and executed by the low level `Acc` task, which is a *SAB* action chooser.

The `Strike` task controls the ball indirectly by means of a collision between the ball and the bat. It requires a model of the real world:

$$\begin{array}{c} \underbrace{\text{Ball at Hit} \times \text{Bat position and direction} \times}_{S} \\ \underbrace{\text{Bat speed}}_{A} \longrightarrow \\ \underbrace{\text{X coordinate of Landed Ball}}_{B} \end{array} \tag{9.12}$$

This too can be learned using a *SAB*-tree. From trial to trial, the `Strike` task attempts to always position the bat to contact the oncoming ball at the bat's centre, and to have it moving in the same relative direction at impact. The speed of the bat at impact is varied. This speed affects the landing position of the ball, and so can be used to indirectly control the landing position. At the trial start the *SAB* action chooser is given $\mathbf{s}_{\text{current}}$ as the estimated ball state when it arrives in range and the bat position and direction during impact. The search produces a recommended speed for the bat during impact.

It should be noted that this *SAB*-tree, like the others, is simply a set of objective observations about the world, and its own accuracy does not depend on the performance of the subtasks which are being learned. There is no blame or credit assignment problem. For example, suppose that we believed that hitting speed S_1 at position P_1 would ensure that the ball landed at X_1, but due to a low level error the ball was volleyed with the correct speed S_1, but at the wrong position P_2. The ball then lands at X_2. The speed S_1 will not now be wrongly associated with landing at X_2: the *SAB*-tree will simply contain an observation that hitting with speed S_1 at the wrong position, P_2, results in a landing at X_2, and will contain no explicit prediction as to what would happen were the ball hit at the correct position.

The `PuckGoal` task guides the initial perceived state to the target impact state. This is ice puck control, again dealt with in the previous section.

9.5.1 Discussion

The volley example has demonstrated the general principle that an apparently complex dynamic control problem can be broken down using naive, qualitative reasoning. The resulting plan uses middle abstraction *SAB*-tasks which can be automated. The interesting observation is that whenever a quantitative relationship is required, *SAB*-learning can be used, and if necessary, the control choice mechanism can be used to invert and explore these other world models.

The state for the lowest-level model, the arm's Perceived State Transition Function, is the raw action for the higher level `Strike` model. This is an entirely natural way to decompose a hierarchical task. An abstract task decides it needs a certain action performed which is delegated to a lower level task. For the lower task, this then becomes the goal state. A particularly pleasing feature of a hierarchy of *SAB*-learners is that they can usefully learn simultaneously, without there being any danger of a credit assignment problem.

9.6 The Benefits of Learning

This chapter has shown how a variety of complexities of robot task can be interfaced with *SAB*-learning. It has discussed the fact that the *SAB* action chooser cannot be used for all aspects of task achievement, but instead some other means must be used to obtain the abstract plans for achieving abstract tasks. However, there are some benefits of *SAB* learning, derived from the raised level of abstraction below which task automation can take place. There are several ways this is achieved:

- It is unnecessary to work in any coordinate system other than that which the robot perceives. Thus, for example, the output of a planner to move a hand towards an observed goal is simply to thrust the observed hand position in the observed direction.

- If it is suspected that the environment has some noise, then it may be unnecessary to take it into account in the generation of the plan. Noise tolerance can be dealt with at the modelling level.

- An environment which might change, either gradually or suddenly, is again dealt with at the modelling level. This is provided that there is no qualitative change, such as part of the robot arm actually disintegrating, in which case a new abstract strategy might be needed.

- If the raw actions can supply sufficiently powerful actions (for example, large enough torques), then the middle levels of control—trajectory following or atomic movements—can be controlled with trivial "ice puck" control.

- World models can be used at multiple locations in the subtask hierarchy. Even if these world models depend upon each other, learning can take place simultaneously with no danger of errors made by one level resulting in other levels learning incorrect models.

There are, however, some extra requirements on the designer.

- It is necessary to know, for each world model, that it *can* be usefully learned as a deterministic, generally continuous function. This does not eliminate noisy functions, but it makes the assumption that the noise distribution is useless and is to be removed.

- It is necessary to supply, along with each requested behaviour, a tolerance of acceptable behaviour. However, no difficulty is foreseen should it be required that these values are generated automatically. Chapter 10 demonstrates that within a wide range, performance is not sensitive to these values.

- In the current implementation, for each *SAB*-tree, it is necessary to provide a small set of parameters. These consists of the constant C used by the P_{succ} estimate, (from Section 8.1) the expected noise level (specified indirectly by the D_{range} search width value of Section 7.2), the number of candidate actions to be considered should the partial inversion action be inadequate, and the spread of values within each dimension of the state, action and behaviour spaces (see Section 5.1). Empirically, we will see that the performance is not sensitive to these parameter choices, nor are they difficult to estimate. It would also be possible to estimate them from the data.

- The current implementation also requires that the *SAB*-tree be occasionally garbage collected (see Section 7.5). The plan must set aside computation time for this purpose. This could also be automated by performing the garbage collection incrementally, or as a background process.

The design and analysis of plans to achieve high level robotic tasks are generally not complex. They are significantly easier than the design and analysis of software systems. Thus, we may hope that robot control based on learned world models makes robot system design a cheap and easy process.

A more interesting alternative is that the planning can be automated. Many researchers have investigated the use of planning for robot tasks. An objection to some of this research has been the *micro-world* problem—that the investigations factored out too much disorder from the environment in order to be able to work at a logical, abstract, level instead of a

numeric, concrete level. But it is apparent that, with a learning modelling system at the low levels of the task structure, such disorder *can* be factored out.

Thus, by learning the world models we have automated the low level part of robot planning. An analogy with biological systems would be to suggest that the SAB models are the equivalent of the low level, subconscious processes of the motor control system, while the task planning is equivalent to a higher level process.

Chapter 10

Experimental Results

This chapter describes the methods, results and conclusions for a variety of learning control experiments. These consist of: learning a hand-eye coordination; learning to follow a perceived trajectory of a simulated torque-controlled arm; learning movement control for non-repetitive tasks; learning to keep a ball bouncing; and learning the volley task described in Section 9.5.

10.1 AB Learning

This section details the results of an experiment performed early on during the research described in this dissertation. As a result, many features of the *SAB* learning system described in this dissertation are missing. However, it is an interesting experiment worth examining because it works with a real robot instead of a simulation and it uses a fairly high dimensional mapping. It is also described in [Clocksin and Moore, 1989].

The experiment learns the combined perspective transformation and the kinematics of a five-jointed robot arm. The robot (an RTX arm manufactured by UMI Inc.) is driven by specifying joint positions, which are then attained by independently driven, servo-controlled actuators. The experimental set-up is shown in Figure 10.1. A camera is pointed at the robot which holds a small pen light in its gripper. The camera is connected to a frame-grabber in which the image is thresholded, distinguishing the bright spot. Some image processing obtained the location of the centre of the bright spot as image x and y coordinates. It was not necessary for the image processing to be sophisticated and so the simple implementation took less than quarter of a second, running on an IBM PC.

The world was learned as a *SAB*-tree:

$$\underbrace{\{\}}_{\textbf{State}} \times \underbrace{\text{Joint Positions}}_{\textbf{Action}} \rightarrow \underbrace{\text{Image Position} \times \text{Comfort}}_{\textbf{Behaviour}} \tag{10.1}$$

The **State** component was empty because the configuration of the system prior to requesting a set of joint positions had no effect on the configuration afterwards. The joint

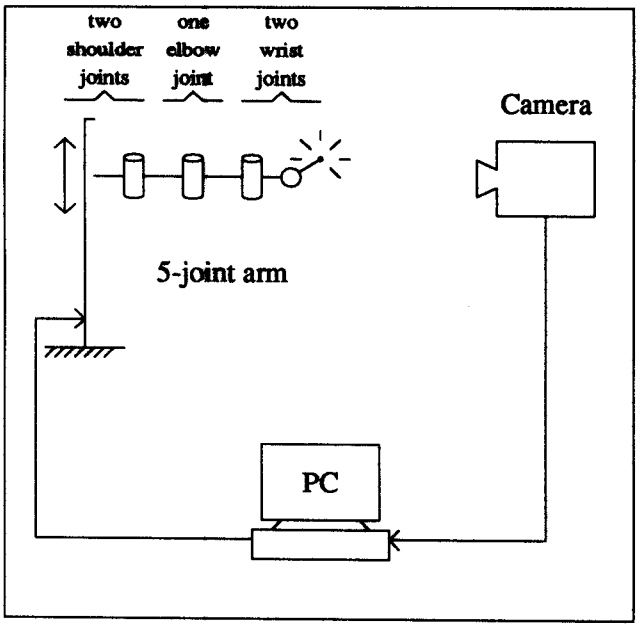

Figure 10.1

The hand-eye coordination experiment.

positions were a vector of five values: shoulder height, shoulder angle, elbow angle, wrist pitch and wrist yaw. Wrist roll was held constant, because some roll configurations obscured the light from the camera. The image position were x and y values specified in pixels. The comfort value was a measure, depending on the configuration of the joints. The closer each joint position was to the centre of its range, the more "comfortable" the joint position. The ranges of all variables were explicitly known. The comfort value was used to choose between alternative joint angle configurations which achieved the same image position. The best possible comfort value would be zero.

The use of "comfort" was a primitive example of task dimension reduction, described also in the experiments of Section 10.4.

In this experiment the *SAB*-tree did no smoothing or adaptation to environmental changes. The *SAB* action chooser was also in a more primitive form, described below.

To test the system a simple game was played in which the robot started with an empty *SAB*-tree. On each trial a random image location was chosen and the system recorded the number of movements which need to be made before the observed image location agreed with the goal image location (to within 12 pixels on a 512 × 512 pixel display).

The prototype action chooser which was used began by using partial inversion to obtain the exemplar which had its behaviour vector closest to the goal behaviour (goal x position , goal y position , comfort = 0). If it was predicted to be good enough then the action was used. Else, a small random perturbation was applied to the recommended joint angles. The result was inspected and stored in the *SAB*-tree. If it had improved upon the pre-

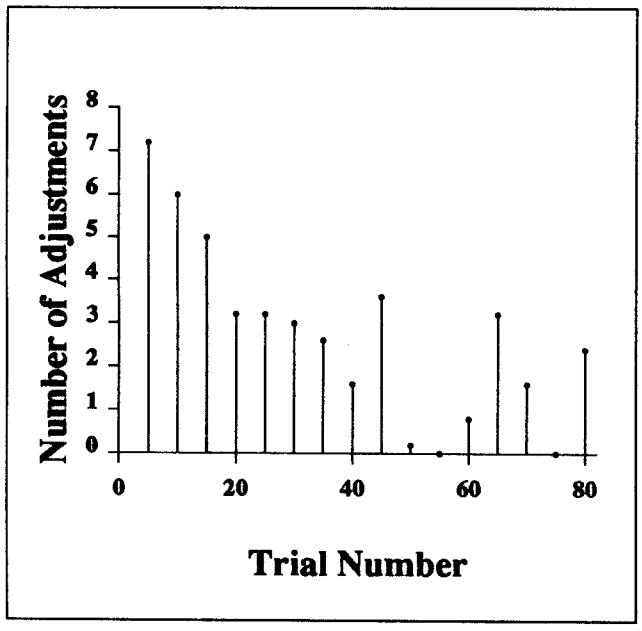

Figure 10.2

Number of extra arm movements required to find the goal image position. The statistics are averages of buckets of five trials.

dicted behaviour it was repeated, else a different random perturbation was selected. This process was repeated until the goal location was obtained. This technique is similar to the Hooke and Jeeves multivariate optimization method [Murray, 1972].

Figure 10.2 shows the performance of the robot for the first eighty such trials, measured as the number of perturbations above the initial movement required to reach each randomly generated target position. The whole experiment, including the rather slow movements of the robot and the image processing, took only 18 minutes.

10.1.1 Local Interpolation

This experiment also used local regression to improve the nearest neighbour generalization. The regression was used in the **Behaviour** $\rightarrow$ **Action** direction of the mapping, by means of the nearest 5 neighbours which surrounded the goal behaviour. It was recognised that this was generally too expensive to be able to perform on-line, anticipating future application of *SAB*-trees to dynamic tasks. To solve this, the interpolation was performed off-line in what were termed "dreaming" periods.

During a "dreaming" period, the software links to the robot and sensors are cut, and replaced by links to the local regresser. The controller continues to learn as if the task were still happening for real, but instead of gaining observations from the real world it gains them from the regresser. The regressed observations are added to the *SAB*-tree, but are tagged as approximations. Once real world learning is switched on again, the controller ensures it verifies any tagged point prior to its use.

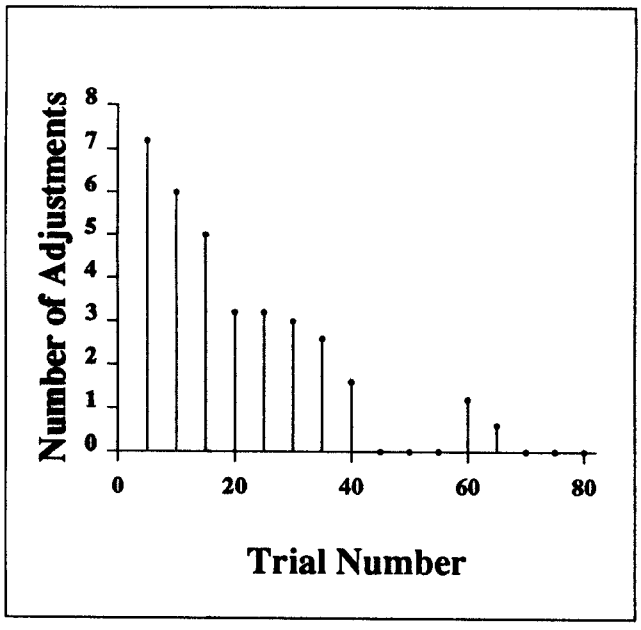

Figure 10.3

Number of extra arm movements required to find the goal image position, with a "dreaming session" just after trial 40. The statistics are averages of buckets of five trials.

Figure 10.3 shows how the performance improved when a dreaming period took place just after the fortieth trial.

The next experiment was to learn to move in sequence through 100 points uniformly spaced on an ellipse specified on the image. First, the robot practised moving to each point in turn for a total of ten minutes. Figure 10.4 shows the results of a "test" given after the practice session: moving as close as it can (according to the *SAB*-tree) to each point. The target sequence of points is between the two white bands shown on the image.

Figure 10.5 shows how it performed on the test with a dreaming period after 10 minutes of practice, but before it verified the memory values regressed during this time. Figure 10.6 shows the results after a further fifteen minutes of practice. The times for these results are relatively short in a field where learning can often take hours or even days.

In the subsequent experiments of this chapter, dynamic control is learned. The RTX robot cannot be controlled in a realistic dynamic fashion and it was therefore necessary to abandon the use of a real robot in favour of simulation.

10.2 SAB Learning—The Two Joint Arm

This section provides a very detailed investigation of one application of *SAB* learning. It is applied to a simulation of the two-jointed dynamic manipulator depicted in Figure 10.7. The arm has the following features:

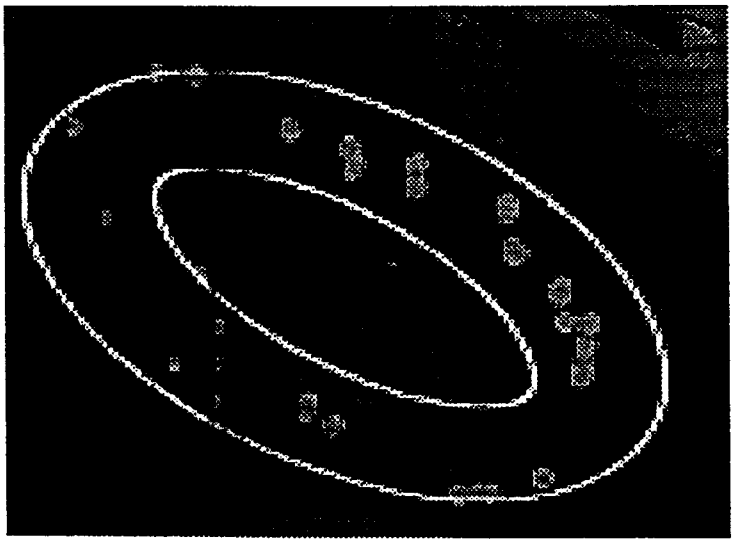

Figure 10.4

The ellipse tracing test immediately after 10 minutes of practice. The bright spots are the positions to which the light was moved. The image of the arm can be seen in the background.

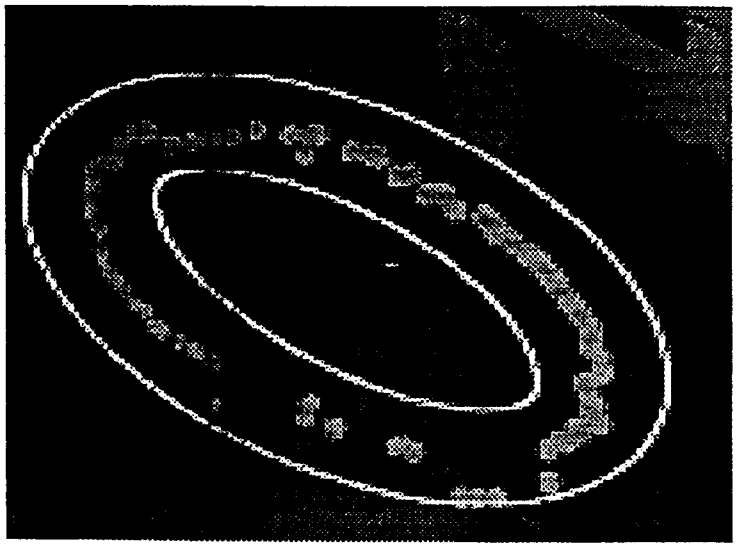

Figure 10.5

The ellipse tracing test after 10 minutes of practice, followed by a period of dreaming but no verification.

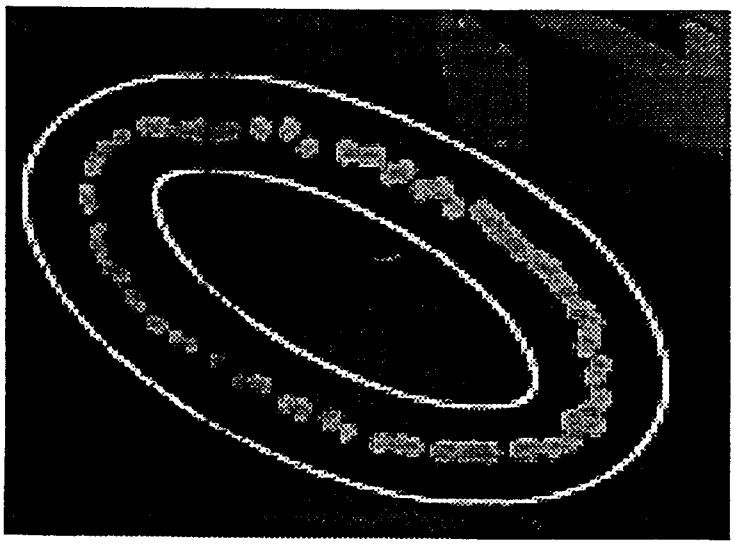

Figure 10.6

The ellipse tracing test after 10 minutes of practice, followed by a dreaming period and followed in turn by 15 minutes of verification and practice.

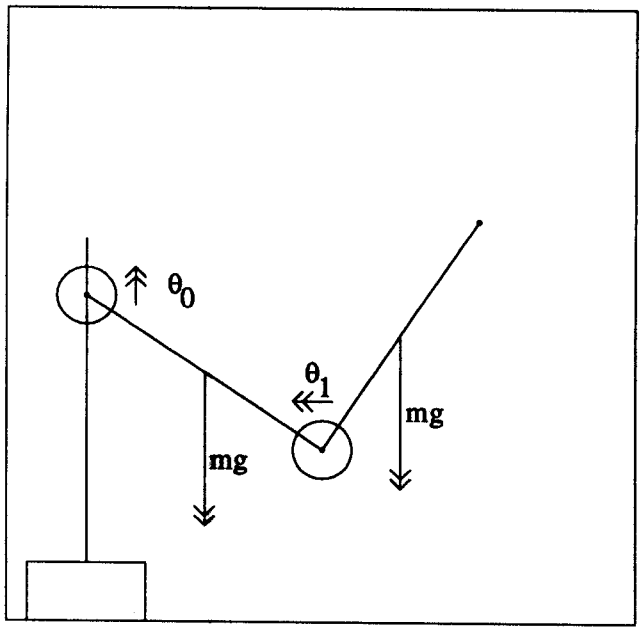

Figure 10.7

A two-jointed dynamic manipulator acting under gravity.

- **Geometry.** The arm is constrained so that it may never straighten: the elbow angle, θ_1, is always between 0 and 180 degrees. The hand may not move outside the frame of the diagram in Figure 10.7.
- **Perception.** The perception is visual. The simulation can, at each time step, view the x-coordinate and y-coordinate positions of the hand. It can also sense the velocity of the hand in both the x and y directions. Because there are only two joints, and because of the constraint that the elbow may not straighten, this is sufficient to define the current perceived state.
- **Dynamics.** It is controlled by two motors: Motor 1 at the shoulder and Motor 2 at the elbow. These each provide a torque. At each time step the torques may be changed.

In this domain the **Perceived State** consists of the observed position and velocity of the hand. The two-dimensional **Action** is the pair of applied joint torques and the two-dimensional **Behaviour** is the perceived acceleration of the hand's position. This is not observed directly, but is estimated as

$$\mathbf{b}_i = \frac{\mathbf{v}_{i+1} - \mathbf{v}_i}{h} \tag{10.2}$$

where $\mathbf{v}_i$ is the perceived velocity on the ith time step, and h is the length of a time step.

The two-jointed manipulator was chosen because (i) it has a six-dimensional control space, which is of non-trivial size (ii) the equations of motion of an idealized two-jointed

SAB-Tree Parameters		*SAB*-Control Parameters		Simulation Parameters	
`Statemax0`	1	`Tolerance`	0.10	`Max-torque`	6
`Statemax1`	1	`Num-cands`	10	`Time-step`	0.02
`Statemax2`	10	`Prob-const`	1	`Noise-level`	0.002
`Statemax3`	10	`Locality`	0.67		
`Actionmax0`	6	`Local-const`	1		
`Actionmax1`	6	`Delta-old`	0.05		
`Bhvmax0`	100				
`Bhvmax1`	100				
`Statemin0`	-1				
⋮					
`Bhvmin1`	-100				
`Search-width`	0.02				

Table 10.1: Arm experimental system parameters

manipulator are well known, and cheap to compute and (iii) manipulators are a particularly useful piece of robot technology. The dynamic equations of motion are from [Fu *et al.*, 1987].

10.2.1 Experimental environment

The simulation and *SAB*-learning were performed in an environment in which the system parameters were all adjustable for experimentation purposes. They are listed in Table 10.1.

`Statemax0` ... `Bhvmin1` are the sixteen values required to scale the vectors in the *SAB*-tree. The need to know the maximum ranges of **State**, **Action** and **Behaviour** components was discussed in Section 5.1. All vectors in the tree are scaled to the range $[0, 1]$. State variables 0 and 1 are the x and y positions respectively. These position maxima were easy to estimate, because the hand may not move outside the coordinates $[-1, 1]^2$. The range of observed speeds (state components 2 and 3) were less obvious but were estimated as 10ms^{-1} from observations of the simulation. The action ranges were again known explicitly, as part of the definition: they are the maximum permissible torques. The behaviour ranges were estimated from observation. The `Search-width` parameter is the range width $D_{\mathbf{range}}$ of the smoothing kernel discussed in Section 7.2.

The *SAB*-control parameters were discussed in Chapter 8. `Tolerance` is the deviation in the behaviour which is deemed acceptable when the *SAB*-controller is requested to generate an action $\mathbf{a}_{\mathbf{raw}}$ to achieve a goal behaviour $\mathbf{b}_{\mathbf{goal}}$. It is scaled to the range $[0, 1]$. `Num-cands` (see Section 8.1) is the number of candidate actions considered when the value recommeded by partial inversion is predicted as inadequate. `Locality` and `Local-const` are the P_{loc} and C_{loc} parameters of Section 8.2, which bias the generation of candidate

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >

Minimal behaviour:    <-10.00,-10.00 >
Maximal behaviour:    < 10.00, 10.00 >
(Empty tree. Random action.)
            Action: < -4.15,  0.37 >
Predicted Behaviour: < NULL VECTOR >
   Actual Behaviour: <-13.39,-37.33 >
```

Table 10.2: Repetitive Control: The first cycle

actions. `Prob-const` (again, see Section 8.1) is the scaling factor C used in the probability of success estimate, P_{succ}. Its value reflects the expected slope of the function being learned, but its accuracy is not expected to be important. `Delta-old` is the Δ_{old} parameter used in the detection of old, inaccurate, exemplars (see Section 7.3).

The simulation parameters `Max-torque` and `Time-step` are self explanatory. The arm is given simulated noise defined by the `Noise-level` parameter. At each stage, the requested torques are corrupted before being applied. The corruption is uniform in each action component, up to a maximum value of `Max-torque` × `Noise-level`.

10.2.2 Convergence of Actions

The first experiment examines how the *SAB* action chooser performs on repeated occasions of the same state and requested behaviour. In this experiment, the state is reset after every control cycle. This hypothetical situation was discussed in Section 8.2, but is unrealistic—normally in subsequent experiments the state, and often the goal behaviour, would vary between cycles. Here, the start state is position and velocity both zero—the hand stationary in the center of the display. The goal behaviour is zero acceleration, which can be achieved by action $(0.98, 0.25)$. The action is non-zero because the torques have to counteract gravity.

Tables 10.2–10.5 consist of *SAB* action chooser debug output. This shows in extreme detail, the behaviour of the algorithm. Table 10.2 shows the performance on the very first cycle. Here, there is no previous experience whatsoever in the *SAB*-tree, and so a uniformly distributed random action is chosen. The minimal and maximal behaviours mentioned in the table are those generated by the default `Tolerance` parameter setting of 0.10 (see above for a description of `Tolerance`).

Table 10.3 shows the performance on the second control cycle. There is one exemplar in the *SAB*-tree, obtained from the first cycle. This is, of course, the exemplar suggested by partial inversion. It is not suitable, because is does not achieve the `Tolerance`. Instead, ten candidate actions are generated (ten, because the `Num-cands` parameter is set to 10 by default). These all predict the same behaviour because they all have the same nearest neighbour. The estimated probabilities of success vary because the reliability of

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >

Minimal behaviour:      <-10.00,-10.00 >
Maximal behaviour:      <10.00,10.00 >
Partial Invert Action: <-4.15, 0.37 >
Nearest BSmooth:        <-13.39,-37.33 > at distance 0.000. Psucc = 0
Cand 0: <-6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.553. Psucc = 0.00910982 (Best yet)
Cand 1: < 6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.999. Psucc = 0.00306144
Cand 2: <-6.00, 6.00 > BSmooth: <-13.39,-37.33 > at distance 0.493. Psucc = 0.0110591 (Best yet)
Cand 3: < 6.00, 6.00 > BSmooth: <-13.39,-37.33 > at distance 0.967. Psucc = 0.00325637
Cand 4: < 1.63,-2.01 > BSmooth: <-13.39,-37.33 > at distance 0.521. Psucc = 0.0100832
Cand 5: < 0.56, 2.34 > BSmooth: <-13.39,-37.33 > at distance 0.426. Psucc = 0.0140408 (Best yet)
Cand 6: < 4.76,-0.07 > BSmooth: <-13.39,-37.33 > at distance 0.744. Psucc = 0.00534009
Cand 7: <-5.60, 0.06 > BSmooth: <-13.39,-37.33 > at distance 0.124. Psucc = 0.0185096 (Best yet)
Cand 8: <-3.34, 3.13 > BSmooth: <-13.39,-37.33 > at distance 0.239. Psucc = 0.0277313 (Best yet)
Cand 9: < 2.49, 1.73 > BSmooth: <-13.39,-37.33 > at distance 0.565. Psucc = 0.0087751
Probability of Success is   0.0277

            Action: <-3.34, 3.13 >
Predicted Behaviour: <-13.39,-37.33 >
   Actual Behaviour: <-84.87, 4.54 >
```

Table 10.3: Repetitive Control: The second cycle

the nearest neighbour prediction varies with distance. Notice that the first four candidate actions are the extreme actions. For this controller they are always generated because they might be needed for the estimation of strength (see Appendix C). The quoted nearest neighbour distances are in the uniformly scaled SAB-tree units. The best candidate action is, unfortunately, not good enough, and the actual behaviour is further from goal than the original.

The third cycle is displayed in Table 10.4. There are now two exemplars, but the original was more promising. As a result it is again returned by partial inversion, but is again, of course, deemed inadequate. Ten further candidate actions are again generated. This time there is variety in the nearest neighbour prediction. Not surprisingly, the most promising candidate has the original exemplar as its nearest neighbour. This time the choice is lucky, and achieves a good behaviour, though it still does not meet the tolerance.

The action discovered by cycle 3 remained the most promising for the next 7 cycles. On trial 10 it was replaced by action $(-1.01, 0.35)$, which produced behaviour $(-6.68, -13.71)$. This was in turn replaced by a slightly closer action on cycle 15. On trial 19, an action was discoved which produced behaviour $(-6.23, 2.40)$—within the tolerance. Cycle 20 is displayed in Table 10.5. All subsequent cycles used the same action. The small mismatch between the predicted and actual behaviour was due to the simulated noise. If there had been no noise the prediction would have been entirely accurate, because the state and action were the same as before.

The whole experiment was repeated for thirty runs. The graph in Figure 10.8 shows,

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >
Minimal behaviour:     <-10.00,-10.00 >
Maximal behaviour:     <10.00,10.00 >
Partial Invert Action: <-4.15, 0.37 >
Nearest BSmooth:       <-13.39,-37.33 > at distance 0.000. Psucc = 0
Cand 0: <-6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.553. Psucc = 0.00910982 (Best yet)
Cand 1: < 6.00,-6.00 > BSmooth: <-13.39,-37.33 > at distance 0.999. Psucc = 0.00306144
Cand 2: <-6.00, 6.00 > BSmooth: <-84.87, 4.54 > at distance 0.326. Psucc = 0.00553616
Cand 3: < 6.00, 6.00 > BSmooth: <-84.87, 4.54 > at distance 0.815. Psucc = 0.00364744
Cand 4: <-1.40, 0.57 > BSmooth: <-13.39,-37.33 > at distance 0.230. Psucc = 0.028409 (Best yet)
Cand 5: <-2.79, 3.06 > BSmooth: <-84.87, 4.54 > at distance 0.046. Psucc = 0
Cand 6: <-0.33, 1.07 > BSmooth: <-84.87, 4.54 > at distance 0.304. Psucc = 0.00495881
Cand 7: <-3.23, 3.54 > BSmooth: <-84.87, 4.54 > at distance 0.035. Psucc = 0
Cand 8: <-4.00, 2.94 > BSmooth: <-84.87, 4.54 > at distance 0.057. Psucc = 0
Cand 9: <-5.75, 4.13 > BSmooth: <-84.87, 4.54 > at distance 0.218. Psucc = 0.00162928
Probability of Success is   0.0284
            Action: <-1.40, 0.57 >
     Goal Behaviour:  [ 0 , 0 ]
Predicted Behaviour: <-13.39,-37.33 >
   Actual Behaviour: <-13.02,-13.96 >
```

Table 10.4: Repetitive Control: The third cycle

```
        Start State: <  0.00,  0.00,  0.00,  0.00 >
Minimal behaviour:     <-10.00,-10.00 >
Maximal behaviour:     <10.00,10.00 >
Partial Invert Action: < 0.90, 0.47 >
Nearest BSmooth:       <-6.23, 2.40 > at distance 0.000. Psucc = 1
Partial Inversion action is tolerable.
Probability of Success is   1.0000
            Action: < 0.90, 0.47 >
Predicted Behaviour: <-6.23, 2.40 >
   Actual Behaviour: <-6.13, 2.08 >
```

Table 10.5: Repetitive Control: The twentieth cycle

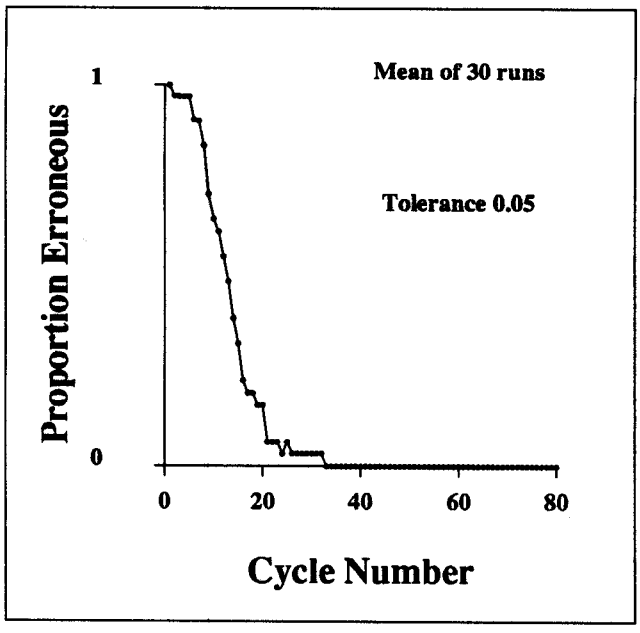

Figure 10.8

Proportion of runs of the repetitive control experiment which met tolerance against cycle number. The tolerance was 10%.

for each cycle number n, the proportion of occasions for which the tolerance not had been achieved by cycle n. Thus, for example, the value is 1 for cycle number 1, because on every run, the action was unsuccessful on trial 1. By cycle 13, 50% of runs had succeeded. This is encouraging, because it agrees with the prediction of Section 8.2, that the correct behaviour is converged to rapidly using the *SAB* action chooser, even with only a few candidate actions. The tolerance was, however, quite large, meaning that approximately 1% of possible behaviours would succeed. How does the time to achieve goal behaviour increase as the tolerance becomes more strict?

A second experiment was performed in which the tolerance was only 0.004, a 25-fold decrease over the previous experiment. The experimental conditions were identical, except that the `Noise-level` parameter was set to zero. This was because the tolerance would not have been possible (except occasionally by luck) with the noise level of the previous experiment. The behaviour is shown in Figure 10.9, and it can be seen to be no more than a small factor worse than the previous experiment. The median number of cycles before achieving tolerance 0.004 was 35.

Further experiments were performed with varying tolerance levels. The results are tabulated in Table 10.6 and plotted in Figure 10.10.

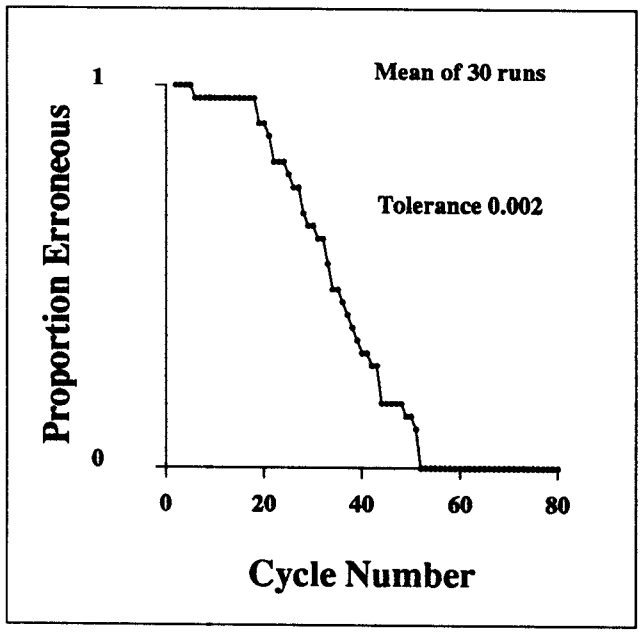

Figure 10.9

Proportion of runs of the repetitive control experiment which met tolerance against cycle number. The tolerance was 0.4%.

Tolerance	Median time to success
0.2	6
0.1	13
0.04	15
0.02	25
0.01	24
0.004	35
0.002	40
0.0004	48
0.0001	58

Table 10.6: Time to achieve tolerance for the simple "holding still" task.

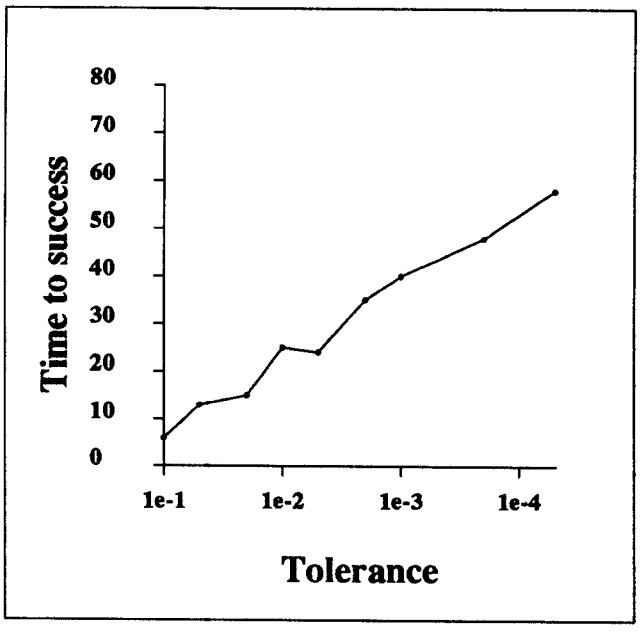

Figure 10.10

A plot of the data from Table 10.6: Median time to find the correct set of torques to achieve the goal tolerance. The horizontal axis is logarithmic.

10.3 Trajectory Tracking Experiments

The trajectory following task is a realistic robot control problem, which will be used to empirically explore the detailed behaviour of the *SAB* control cycle. I will begin this section by describing the experiment in detail, and will then explore the effects of varying the following features:

- Number of candidate actions.
- Tolerance level supplied to the *SAB* action chooser.
- Locality C_{loc}, and smoothness estimate C.
- Weighting of variables in the *SAB*-tree.
- World noise.
- Environmental changes.

10.3.1 The Trajectory Tracking Task

The experiment can be viewed at the *trial* level, the *learning run* level and the *experiment* level.

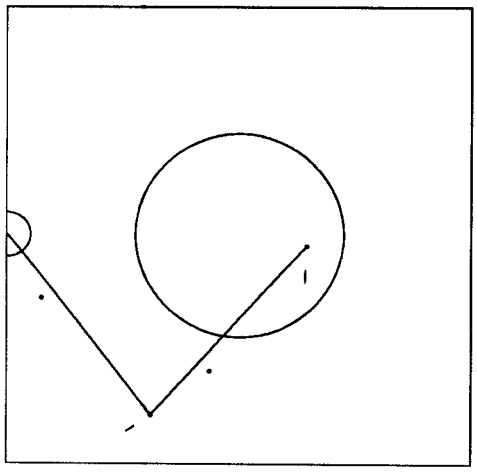

Figure 10.11

The simulated arm must follow the circular trajectory in an anticlockwise direction. It starts from the rightmost point. Here, it is shown during the end of an initial, unsuccessful, learning trial.

- **Trajectory Tracking Trial.** The arm must follow a perceived circle at a specified speed. The circle is shown in Figure 10.11. The goal speed takes fifty simulated time steps to travel around the circle. The abstract level controller uses the ice puck trajectory tracking mechanism of Section 9.3. The hand is initialized at the right-most point of the circle. Then on each cycle, the goal behaviour is the ice puck acceleration required to stay as close to the current trajectory as possible. One trial consists of one attempt to follow the circle.

- **Trajectory Tracking Learning Run.** A run consists of thirty-two trials. Before the initial trial, the *SAB*-tree is empty—the controller begins with no world knowledge. Between trials the world knowledge is maintained, and so if learning occurs, performance should tend to improve during a learning run.

- **Trajectory Tracking Experiment.** The experiment consists of executing a number of independent learning runs. The purpose of running a number of times is to obtain more reliable statistics about the learning performance. There are typically 20 runs in each experiment.

10.3.2 Format of Results

The results for the first experiment are shown in Figure 10.12. However, before they can be discussed it is necessary to explain how they were obtained and what information they are intended to convey.

The **Mean Tolerance** graph plots the behaviour of the *SAB* action chooser against trial number. The *tolerance error* is the amount by which the actual behaviour differs from the tolerance band defined by the task. Each trial has a mean tolerance error. Thus for

each learning run a graph of mean tolerance error against trial number can be obtained. The graph in the figure shows the average of 23 such graphs, obtained from 23 learning runs. The vertical bars display the standard deviations of the statistics. Notice that an experiment in which the mean tolerance error is low is not necessarily performing the task well—it is simply achieving the tolerance. If a large tolerance were specified then it would be easy to achieve a zero mean tolerance error. The axis is labelled as a percentage of the maximum possible tolerance error.

The **Mean Position** graph plots the average position error against trial number. At each time step the position error is computed as the Euclidian distance between where the hand is, and where it should be. The average error for each trial is recorded. The graph is constructed as the average of the 23 observations for each trial. This is a more direct measure of the performance of the task—it would only ever be zero if the trajectory were followed precisely. It is labelled as a percentage of the maximum possible position error, which would occur if the desired position were in the bottom right of the display and the hand were in the top left.

A learning system which learned to perform 98% of the trajectory perfectly, but consistently failed badly at the remaining 2% would be judged well by the Mean Position estimate. For some uses of trajectory tracking however, such performance would be inadequate. This undesirable behaviour would be detected by the **Worst Position** graph which plots the average of the worst position errors which occur on each trial. The *worst position error* of a trial is the furthest Euclidian distance which the hand reaches from the desired point on the trajectory.

The graphics on the right of the diagram display six examples of the trajectories followed by the arm during learning. These are all taken from the same randomly chosen experiment. The edge of the gray disc indicates the desired trajectory. The graphics display the performance on the 1st, 2nd, 4th, 8th, 16th and 32nd trial. They are shown in order to provide a more intuitive feel of the performance of the arm, in the absence of an actual animation of its behaviour. They cannot be used reliably to compare between different experiments because (i) they are only random examples, and so may not be typical of the performance of the experiment, particularly when there is high deviation between runs and (ii) they are only shown with limited resolution, and so apparently circular motion on the diagram might still contain tracking errors. A particular kind of undetectable error would be the case where the hand lags behind the desired position.

The small table of data in the bottom right hand corner gives the following information:

- The experiment name, "BASIC", named as such because it is this basic experiment to which others will be compared.
- The number of runs in the experiment. The mean behaviours displayed by the graphs on the right become more reliable as indicators of the true mean behaviours, the greater the number of runs. For 23, the mean behaviour is, with confidence

95%, estimated to lie within 0.5 sample standard deviations of the sample mean (as mentioned earlier the standard deviations are represented by the vertical bars—see Appendix A for further details).

- The tolerance level used by the *SAB* action chooser. The units are as a proportion of the estimated minimal and maximal ranges of behaviour. A high value is easier to achieve, but may lead to reduced task performance.

- The *Learned By* statistic. This figure is a score which will be used to quantitatively compare different experiments. It is the number of the final trial upon which the mean tolerance error exceeds a small fixed threshold (4%). This threshold is indicated near the bottom of the y-axis of the Mean Tolerance graph. The *Learned By* statistic will be used to compare the performance of the *SAB* action chooser with different parameter sets.

- The *Final Position Error* statistic. This is another score, used to quantitatively compare the overall task performance of different experiments. It consists of the mean position error during the last six trials of each run: it is thus the mean of the final six values of the Position Error graph.

Neither the *Learned By* statistic, nor the *Final Position Error* statistic are particularly useful in isolation, but they will prove useful as comparators between experiments.

10.3.3 Discussion of the BASIC experiment results

In this experiment all the learning parameters were the default values given in Section 10.2. The results demonstrate that *SAB* learning does indeed learn to control this task. By the standards set above it took only eight trials to, on average, achieve adequate performance, and after sixteen trials the tolerance error was always almost zero, indicating that on all trials, the tolerance was almost always achieved.

The task was also learned accurately. In the later trials, the position was typically at worst no more than 0.5% away from the desired distance. Good positional tracking was obtained early on–after only the 4 trials the average position error was only 2%. The sample standard deviations after the first 10 trials are low, reflecting the confidence that the learning runs were not simply lucky.

Figures 10.13–10.16 show the distribution of exemplars in the *SAB*-tree after a learning run. Some components, particularly the position and velocity components are clustered around the important areas of state space. This is a desirable result, demonstrating the variable resolution available from nearest neighbour learning.

Each trial took only 20 seconds to simulate, learn and display graphically. The parameter values used in this experiment were not chosen with great care—they were the *SAB* learning system defaults. The exception to this were the state, action and behaviour minimum and maximum ranges which were estimated as what seemed reasonable after playing

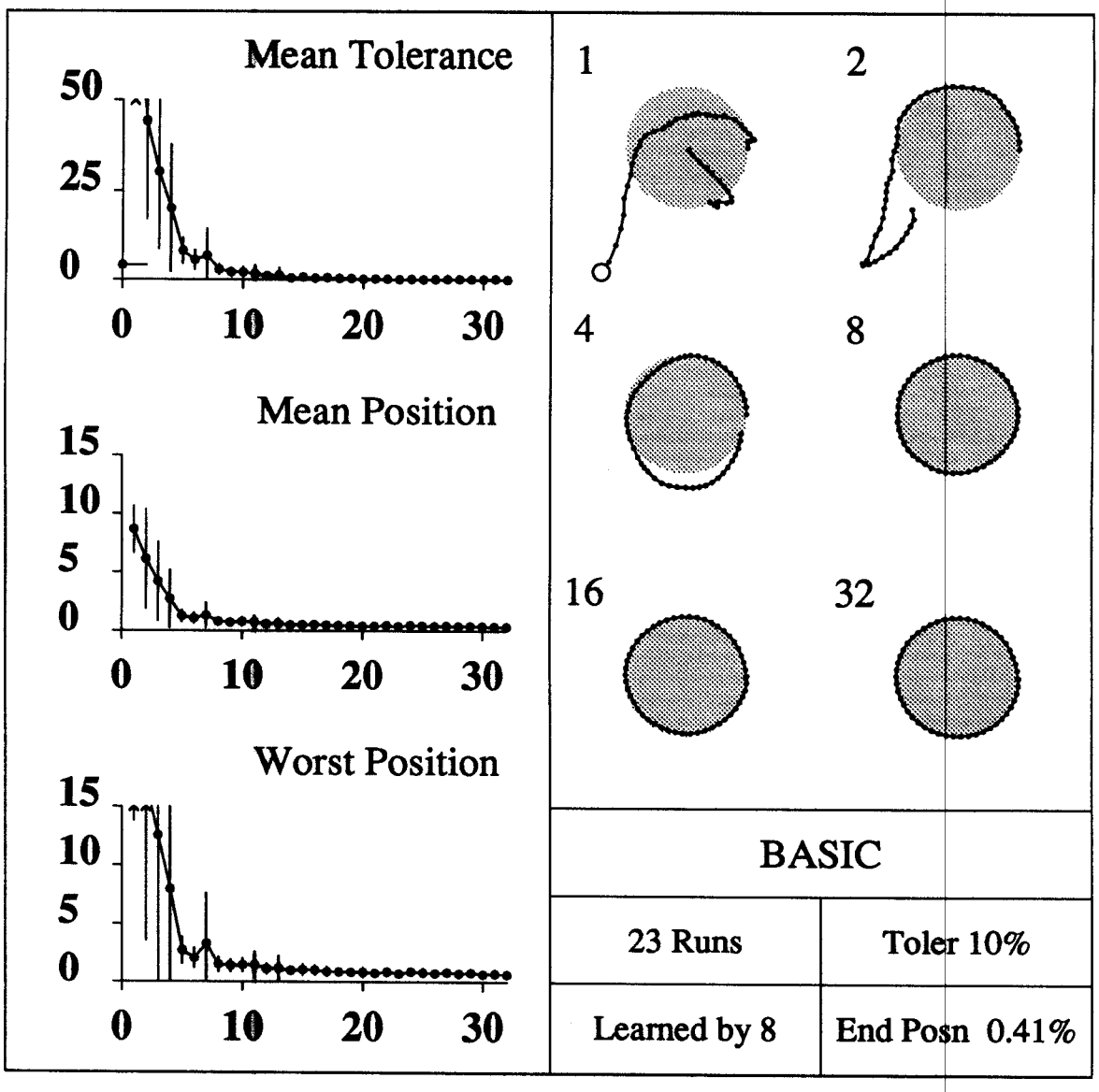

Figure 10.12

Experimental method described in the text. All parameters were default.

with the simulation for a short time. It is nevertheless important to determine empirically whether the system is brittle as the values change. It would also be interesting to see if the performance improves with other parameters. Such experiments are documented in the following subsections.

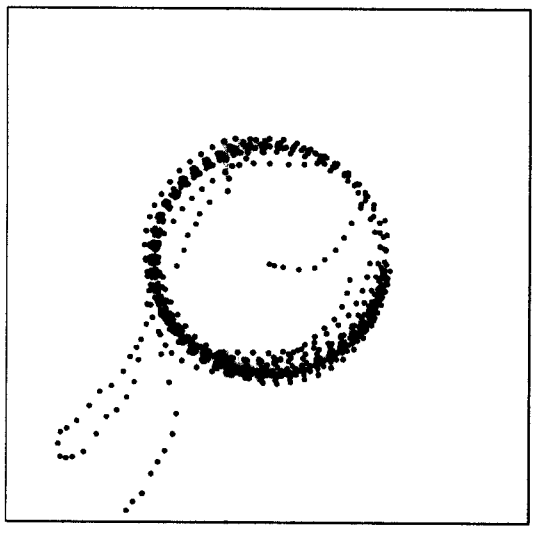

Figure 10.13: The distribution of exemplars in "position" space after a learning run of the BASIC Experiment.

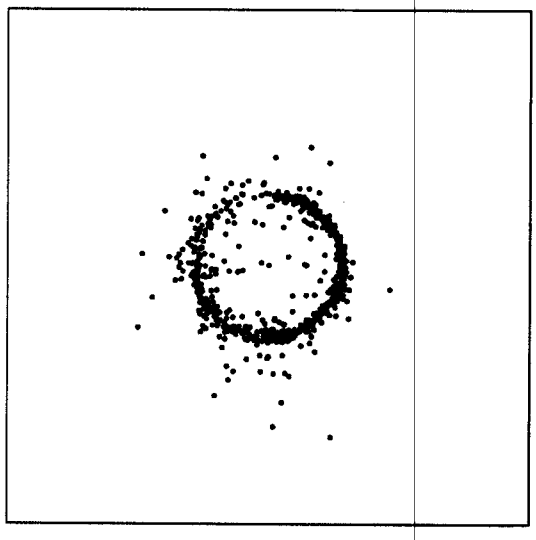

Figure 10.14: The distribution of exemplars in "velocity" space after a learning run of the BASIC Experiment.

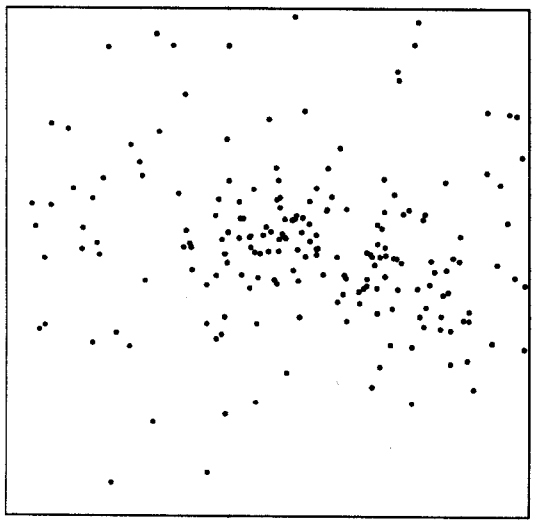

Figure 10.15: The distribution of exemplars in "action" space after a learning run of the BASIC Experiment.

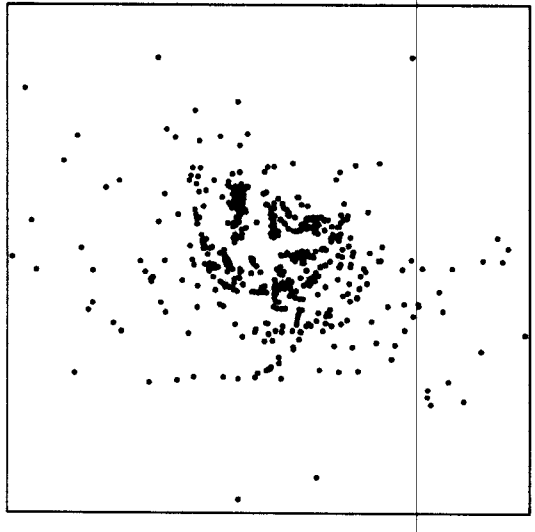

Figure 10.16: The distribution of exemplars in "behaviour" (that is, acceleration) space after a learning run of the BASIC Experiment.

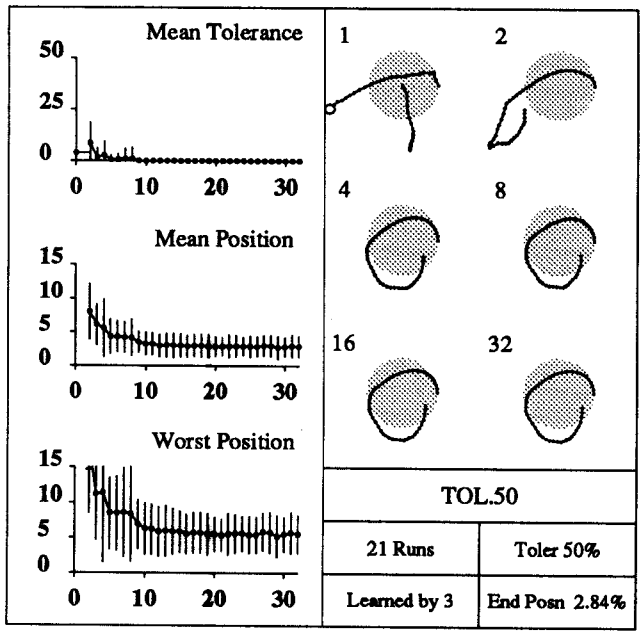

Figure 10.17

Experimental method described in the text. All parameters were default with the exception of the `Tolerance` parameter, which had value 50%.

10.3.4 Tolerance

Making the `Tolerance` parameter larger makes it easier to learn, but means that the accelerations (behaviour) which are supplied might be further from those recommended by ice puck control. An extreme example of this is the TOL.50, experiment displayed in Figure 10.17 in which any behaviour within half the maximum range of behaviours to the goal was considered acceptable. Unsurprisingly this learns to achieve the requested tolerance very quickly, but the performance is poor. The final mean position error of 2.84% corresponds to an average 8cm deviation from the trajectory. There is very little diversity in the actions—in practice the same action is repeated many times for many parts of the state space. This results in the smooth, but wrong, curves of the top half of the trajectory. Very little changes between trials 4–32 because there is no need to refine the behaviour. The results for a variety of experiments with other tolerances are summarized in Table 10.7. Each experiment had identical parameters to the BASIC experiment, except for the `Tolerance`.

The pattern is clear that with decreasing tolerance size the time to learn increases and the final performance accuracy improves. The final experiment, with a tolerance of 0.2% was too accurate to be achieved in only thirty-two trials. It is possible that because of the simulated noise it would never have been achieved. However, there is a large range of `Tolerance` values for which the learning is both fast and accurate, and so it has, fortunately, proved not to be a critical parameter. It is notable that its ideal value

Tolerance	Learned By	End Position Err
50%	3	2.84%
30%	5	2.17%
20%	6	0.73%
10% (BASIC)	8	0.41%
4%	21	0.35%
2%	25	0.33%
0.2%	Didn't Learn	0.64%

Table 10.7: Trajectory tracking: Performance variation with **Tolerance**

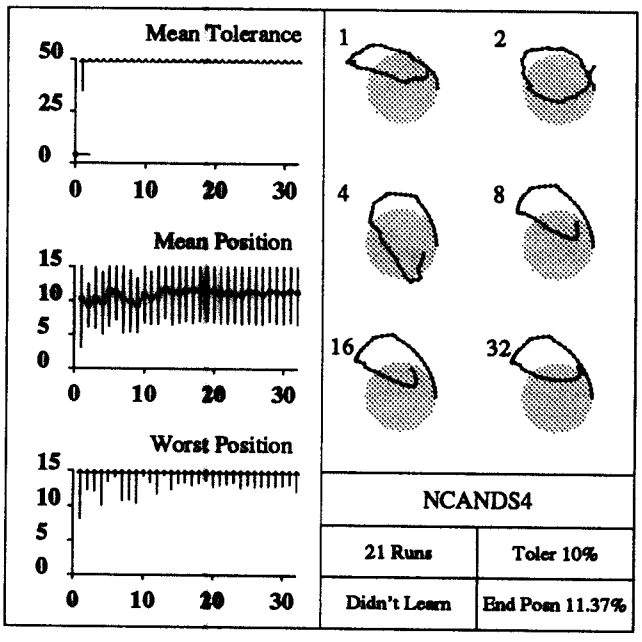

Figure 10.18

Experimental method described in the text. All parameters were default with the exception of the **Num-cands** parameter, which had value 4.

depends on the abstract task being performed. Many tasks do not require high accuracy in their trajectory tracking, and so tolerances as large as 50%, with their impressive rate of learning, might be desirable.

10.3.5 Number of Candidates

At each control cycle, in the absence of a known best action, a number of candidate actions are considered. The BASIC experiment used 10 candidates. At least four candidates, the extreme actions are always considered. Figure 10.18 shows the result when these are the *only* candidates considered. The performance is consistently bad, indicating that the task cannot be achieved by extreme actions alone.

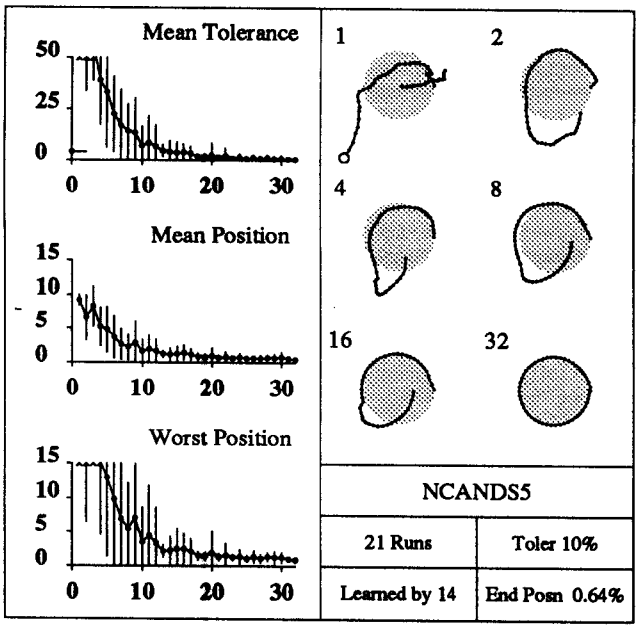

Figure 10.19

Experimental method described in the text. All parameters were default with the exception of the **Num-cands** parameter, which had value 5.

Figure 10.19 displays an experiment with five candidate actions. The performance is considerably improved, but learning is still much slower than that of the BASIC experiment. The final position error is also inferior. This experiment is equivalent to a combination of 'SR'- and 'SL'-control discussed in Section 8.1 in which in the absence of known success a random action is taken.

The other extreme is shown in Figure 10.20 in which 100 candidate actions are considered on each cycle. The learning speed is faster than for the BASIC experiment. After four experiments, when the tolerance is achieved, no further performance improvement takes place because the *SAB* action chooser needs not experiment any further. The real time to run the 100-candidate experiment was about five times greater at the start than the BASIC experiment, but the total time was approximately the same. This is because once success has been achieved, the hundred candidates do not need to be generated—the partial inversion action is always suitable.

Table 10.8 displays the performance of experiments with various values of the **Num-cands** parameter.

The importance of having more than a small number of candidates is demonstrated more clearly when the experiments are conducted with a very small value of **Tolerance**. Table 10.9 displays the performance of the system when the **Tolerance** is 0.2%. The 50 candidates experiment achieved the best final position error seen so far. This is because of the combination of aiming for behaviour very close to that desired by the ice puck controller and having sufficiently many candidates that this accuracy is achieved within

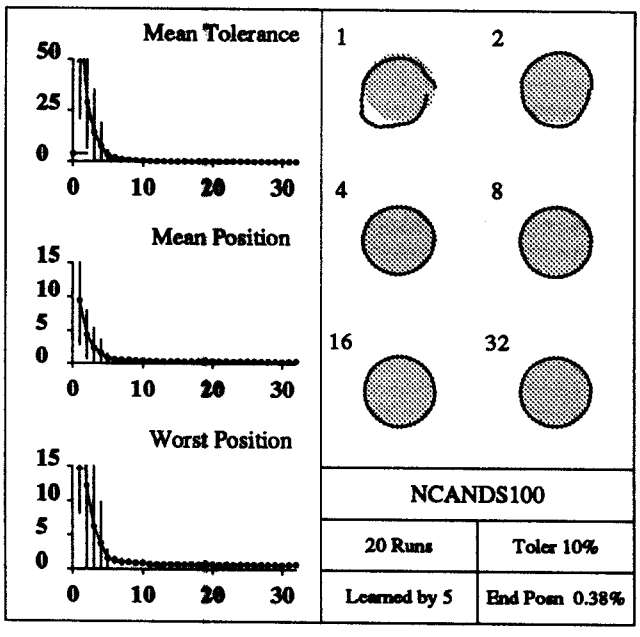

Figure 10.20

Experimental method described in the text. All parameters were default with the exception of the **Num-cands** parameter, which had value 100.

Num-cands	Learned By	End Position Err
4	Didn't Learn	11.37%
5	14	0.64%
6	13	0.45%
10 (BASIC)	8	0.41%
50	6	0.39%
100	5	0.38%
200	4	0.41%

Table 10.8: Trajectory tracking: Performance variation with number of candidate actions

Num-cands	Learned By	End Position Err
5	Didn't Learn	1.43%
10 (BASIC)	Didn't Learn	0.64%
50	24	0.17%

Table 10.9: Trajectory tracking: Performance variation with **Num-cands** with a very small Tolerance value (0.2%).

Locality	Learned By	End Position Err
0	29	0.93%
0.67 (BASIC)	8	0.41%
1	9	0.39%

Table 10.10: Trajectory tracking: Performance variation with **Locality**

the space of 32 trials.

10.3.6 Other *SAB* action chooser Parameters

The **Locality** (P_{loc}) and **Local-const** (C_{loc}) parameters influence the generation of candidate actions. As discussed in Section 8.2, a candidate is chosen local to the best known action with probability **Locality**. If it *is* generated locally, then it is generated within a sphere which has radius proportional to the error of the current best known behaviour. The constant of proportionality is **Local-const**. The experiment with **Locality** of zero (in which candidates are always entirely random) is shown in Figure 10.21. The learning rate is significantly slower than that of BASIC. With **Locality** set to one the experimental results were almost identical to BASIC. These results are summarized in Table 10.10. The possible advantage of choosing a **Locality** less than one is not demonstrated here. This is the fear that only using local improvements might lead to local minima in the control choice.

The behaviour with the **Local-const** appears in Table 10.11. A low value of 0.1 slows down the learning because the local changes made are too small—the P_{succ} heuristic would score candidates higher which were further away, but with such a low **Local-const** such candidates are rarely generated. Values of **Local-const** between 0.5 and 3 do not greatly differ from the performance of the BASIC experiment.

The **Prob-const** parameter, C, alters how the P_{succ} heuristic judges the unreliability of nearest neighbours of differing distance. A low **Prob-const** causes the heuristic to score candidate actions as low if they are at a significant distance from the nearest exemplar. A high **Prob-const** treats the nearest neighbour estimate as reliable even if it is at a large distance. The results of a very low value (0.01) are shown in Figure 10.22. They are very bad! This is because nothing of any significant distance from the best known action is ever tried, and as a result the same action is repeatedly used throughout the entire learning

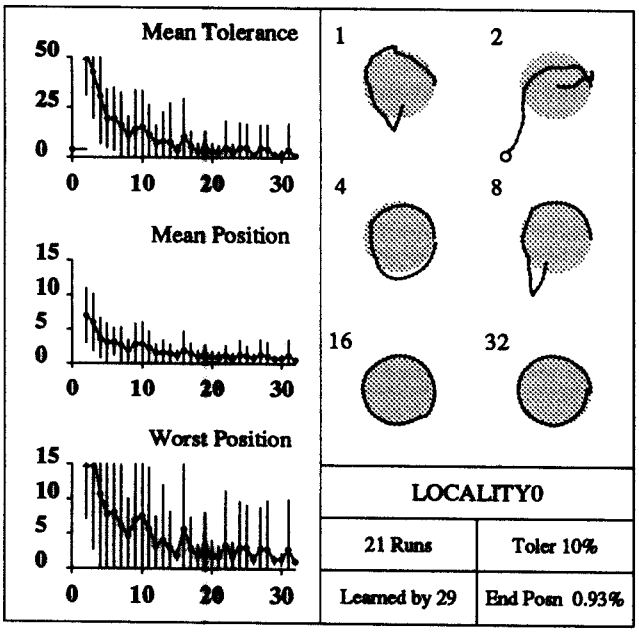

Figure 10.21

Experimental method described in the text. All parameters were default with the exception of the `Locality` parameter, which had value 0.

`Local-const`	Learned By	End Position Err
0.1	18	0.67%
0.5	6	0.35%
2 (BASIC)	8	0.41%
3	15	0.45%

Table 10.11: Trajectory tracking: Performance variation with `Local-const`

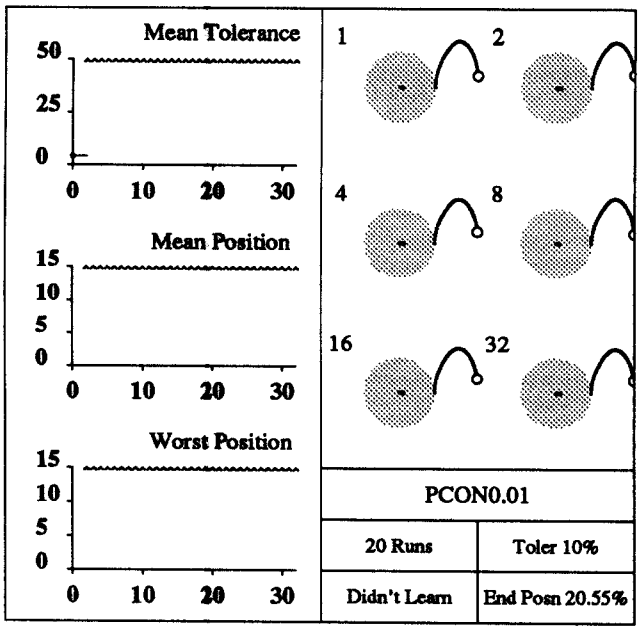

Figure 10.22

Experimental method described in the text. All parameters were default with the exception of the `Prob-const` parameter, which had value 0.01.

`Prob-const`	Learned By	End Position Err
0.01	Didn't Learn	20.55%
0.1	9	0.38%
1 (BASIC)	8	0.41%
2	9	0.49%
5	15	0.8%
10	Didn't Learn	1.88%

Table 10.12: Trajectory tracking: Performance variation with `Prob-const`

run. A very high value (10), displayed in Figure 10.23, does improve, but takes longer than BASIC. The results of other values, tabulated in Table 10.12, indicate that performance is not altered greatly for values between 0.1 and 5.

10.3.7 *SAB*-tree parameters

The *SAB*-tree parameters are the relative weights given to the components of the state, action and behaviour vectors. In the BASIC experiment they were chosen roughly based on the observation of the maximum values of each aspect of behaviour. Four experiments were performed with different weight values. The weightings and results are tabulated in Table 10.13. For example, the WEIGHT.ALPHA experiment was told that the maximum possible perceived x-coordinate of the hand was 3m (BASIC had been told correctly that

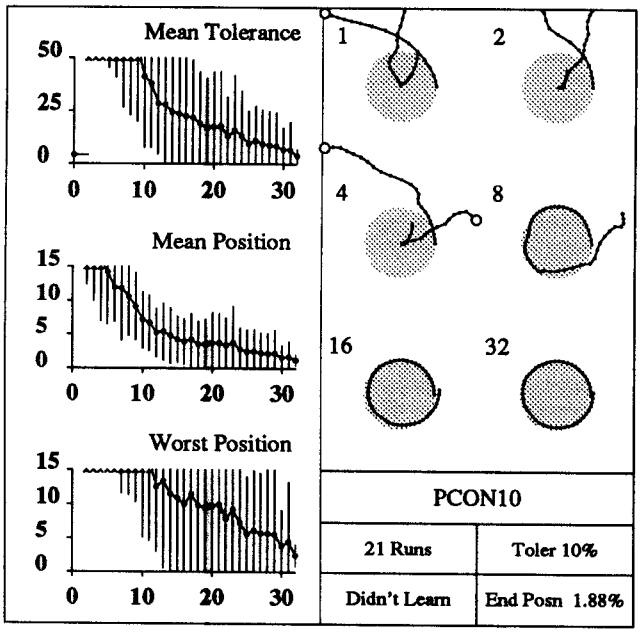

Figure 10.23

Experimental method described in the text. All parameters were default with the exception of the `Prob-const` parameter, which had value 10.

Experiment	Maximum Components			Learned	End Position
	State	Action	Behaviour	By	Err
BASIC	(1,1,10,10)	(6,6)	(100,100)	8	0.41
WEIGHT.ALPHA	(3,1,30,10)	(6,6)	(300,100)	6	0.59
WEIGHT.BETA	(3,3,30,30)	(6,6)	(100,100)	13	0.44
WEIGHT.GAMMA	(1,1,10,10)	(6,6)	(300,300)	7	1.74
WEIGHT.DELTA	(3,3,10,10)	(6,6)	(100,100)	7	0.37

Table 10.13: Trajectory tracking: Performance variation with a variety of weight sets

it was 1m). It was told that the maximum x-direction speed was three times greater than that of BASIC and it was similarly told that the maximum x-direction behaviour (i.e. acceleration) was three times greater than that of BASIC. As a result the nearest neighbour prediction was more sensitive to the y-direction. This is illustrated in Figure 10.24, in which the position components of the vectors stored in the *SAB*-tree are displayed.

The altered weighting can be seen to not have a great effect on performance. The reduced performance in the WEIGHT.GAMMA experiment is because the `Tolerance` is obtained from the scaled space of Behaviours. Thus, a tolerance of 10% when the maximal behaviour is 300 is equivalent to a tolerance of 30% when the maximal behaviour is 100.

The weights of the actions were not altered for any of the experiments. This is because their alteration would have also altered the strength of the arm.

The other *SAB*-tree parameter is the range width D_{range}. This parameter is called

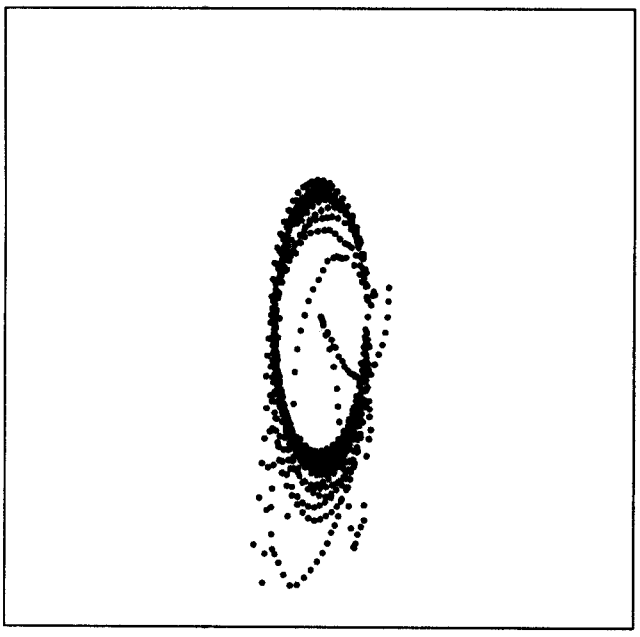

Figure 10.24

The distribution of exemplars in "position" space after a learning run of the WEIGHT.ALPHA Experiment.

Search-width	Learned By	End Position Err
0.002	9	0.49%
0.02 (BASIC)	8	0.41%
0.06	11	0.44%
0.12	12	0.48%
0.2	Didn't Learn	0.6%

Table 10.14: Trajectory tracking: Performance variation with `Search-width`

`Search-width`. The simulated noise is significant but not very large compared with the goal tolerance in the BASIC experiment and so a smoothing kernel of a lower width than the default can be expected to suffice. However, it is interesting to consider the result of increasing the smoothing kernel.

The results are shown in Table 10.14. They indicate that even with a large smoothing kernel, performance is not badly affected. However, the largest kernel, with width 0.2 of the scaled *SAB*-tree width never learns, despite achieving a fairly good final error position. This is because the very strong smoothing makes a constant regular error (similar to smudging an image). As a result the behaviour predictions have a regular constant error. In this case the error is fortunately not large enough to significantly impair the position.

The experiment with a search width of 0.2 took approximately 10 times longer than the BASIC experiment. This is because of the large range search required for each update of the *SAB*-tree.

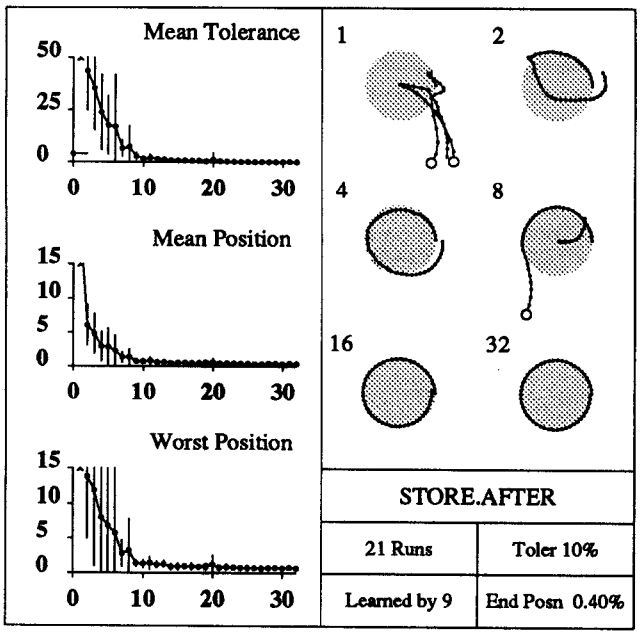

Figure 10.25

Experiment in which the *SAB*-tree is only updated after each trial instead of during each trial. Note the appalling behaviour in trial one—no knowledge of the world is available for the entire trial.

10.3.8 Learning only after Execution

In this experiment I consider the performance if the computation involved with storing new exemplars is taken out of the *SAB* control cycle, and is performed off-line. Trajectory tracking falls into the class of task in which there are natural periods in which extra processing can take place—namely before each trial occurs. This experiment uses this period of time to update the *SAB*-tree with all the data observed in the previous trial. Thus learning does not occur *during* each individual task execution. The results are in Figure 10.25. Performance is reduced, compared with the performance of the BASIC experiment, but not very badly. This result might be useful if the expense of updating the *SAB*-tree were too large to perform within the timescale of the robot's dynamics.

10.3.9 Approximating the Nearest Neighbour

The nearest neighbour search described in Chapter 6 performs a branch and bound search of the *k*d-tree. An objection to this is that although computation is expected to be fairly cheap ($O \log N$), it is not guaranteed. A partial remedy, suggested in [Omohundro, 1987] is to perform normal nearest neighbour search, but after a fixed number, `Max-ninsp`, of nodes have been inspected, to halt the search with the nearest neighbour found to date.

This method was tested for `Max-ninsp` values of 1, 2 and 5. When `Max-ninsp` = 1 the search consists simply of finding the leaf node containing the query point. The results are given in Table 10.15. They show that learning does still occur well, with only slight

Max-ninsp	Learned By	End Position Err
1	13	0.52%
2	11	0.55%
5	10	0.50%
∞ (BASIC)	8	0.41%

Table 10.15: Trajectory tracking: Performance variation with **Max-ninsp**

Search Width	Final Position Error	
	Noisy	Very Noisy
0.2%	1.45%	6.0%
2%	1.22%	6.7%
6%	0.93%	5.52%
12%	0.76%	3.12%

Table 10.16: Trajectory tracking in a noisy environment: Performance variation with Search-width

degradation against the BASIC experiment.

The real time to perform the experiments was not noticeably less than for experiments involving normal nearest neighbour search. A disadvantage of approximating the nearest neighbour is the increased difficulty of analysis of the learning method and some danger of erroneous predictions becoming stuck.

10.3.10 Noise Tolerance

In this section the amount of noise in the environment is increased. We then inspect the extent to which the smoothing of Section 7.2 can help improve the performance. The noise in the BASIC experiment was produced by corrupting the requested torques by a random amount up to 0.2% of the maximum torque values. This can be observed to produce unpredictable behaviour corruptions of up to 1%. This default noise level is not severe, and does not impact greatly on the performance of the system. In eight experiments with much greater noise the smoothing kernel width was varied to discover how much effect it had on combating noise. The performance could never be expected to reach that of the noise free system, because the unpredictable noise was always being added. The controller has a similar problem to a person trying to do neat handwriting on a bumpy train journey.

The noisy environment had the action corrupted by up to 5% and the very noisy environment by up to 20%. The results are in Table 10.16. None of the above experiments achieved a "Learned By" value. This is predictable because the environment is so noisy. There is a noticeable improvement in the final position error as the search width is increased, but it is not sufficient to be classed as achieving the task well. Figure 10.26 shows the performance in the very noisy environment with a low search width, and Figure 10.27

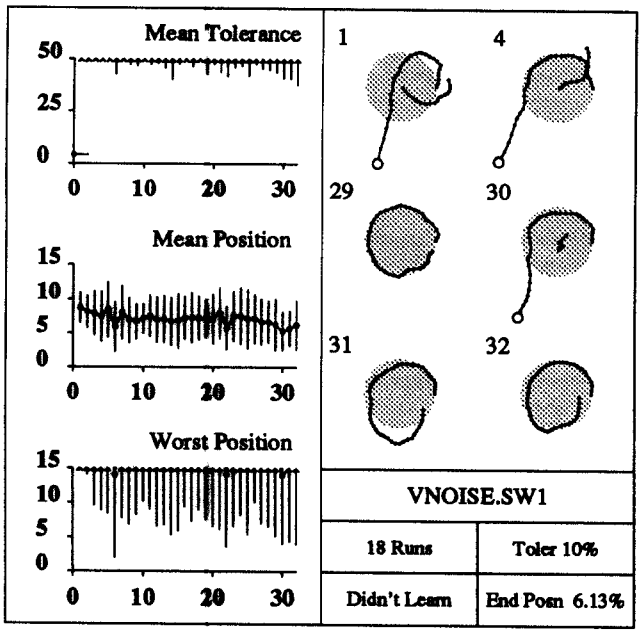

Figure 10.26

Experimental method described in the text. All parameters were default with the exception of the `Search-width` parameter, which had value 0.2%, and the `Noise-level` parameter which had value 20%.

shows performance with a large search width. The final animations were taken without noise. This helps indicate how accurate a model of the noiseless world the system has after 30 noisy trials.

10.3.11 Non-stationary Environment

Recalling Section 7.3, the *SAB* learning controller should be able to adapt to changes in the environment by noticing that old points are inaccurate and deleting them. Here, we test this behaviour. The standard BASIC experiment is used, except that unknown to the controller, just before trial 11, the arm's dynamics are changed severely. The motor for the elbow starts producing $1\frac{1}{2}$ times the requested torque. The results of Figures 10.28 and 10.29 show the effect on performance. The animations correspond to different trial numbers from the earlier figures. Trial ten, which is the last trial before the change, is shown first. A selection of subsequent trials are also shown, and indicate the behaviour after the change, both in the long and short terms. Figure 10.28 shows the effect with change adaptation switched off—points are never thrown away no matter how old or inaccurate they are. Thus, after trial 10, mistakes can be seen to be made for the rest of the run. With adaptation switched on (Figure 10.29), using a `Search-width` parameter of 6%, adjustment performance can be seen to be superior, with an eventual return to acceptable performance. The return to improved performance is, however, not as fast as was the original learning which started from no experience. It is likely that there are two compounded reasons for this.

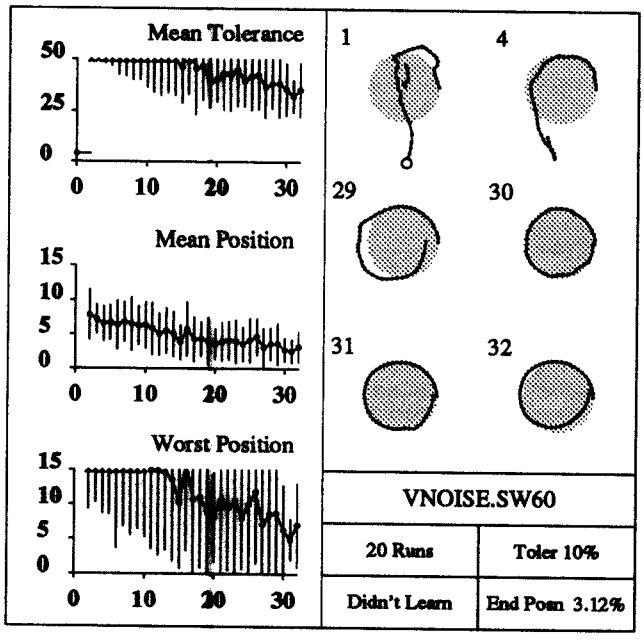

Figure 10.27

Experimental method described in the text. All parameters were default with the exception of the `Search-width` parameter, which had value 12%, and the `Noise-level` parameter which had value 20%.

1. The environmental change is very large, meaning that the new PSTF is, to a large extent, an entirely different function.

2. The learning controller is not warned that a change has occurred, and so must have time to observe this.

From these results it could be argued that the simpler scheme of forgetting everything after a period of increased average prediction error would be preferable. The objection to this is that real environmental changes are likely to (i) be much less severe and (ii) have an affect only in a limited portion of the state space, meaning a loss of valuable data if all information is removed.

10.3.12 Performance of the *k*d-tree

The real-time performance of these learning trials was observed to be good: typically 20 seconds to simulate, learn and display. This gives indirect evidence that nearest neighbour search is performing as desired. To test this more directly the search algorithm was monitored for how many Euclidean distance measurements were taken. These measurements are the dominant cost in nearest neighbour search—we can be sure that the cost of reaching the leaf node in the tree is logarithmic (since the tree is balanced whenever the depth exceeds a small constant times the logarithm of the number of nodes in the tree). Furthermore, reaching the leaf is cheap because Euclidian distance calculations are not required. The details of the nearest neighbour search algorithm are given in Section 6.4.

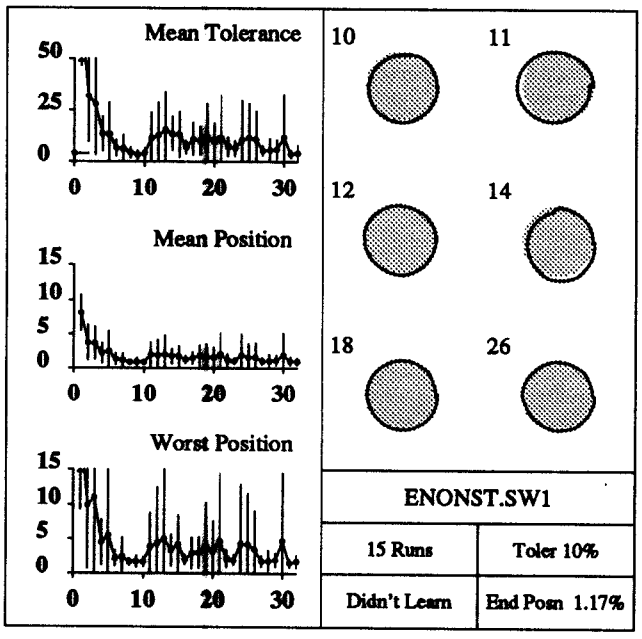

Figure 10.28

A non-stationary environment. The arm dynamics change on trial 10 to produce 1.5 times the torque that they used to. This experiment has a very small search width, meaning very few old inaccurate exemplars are killed.

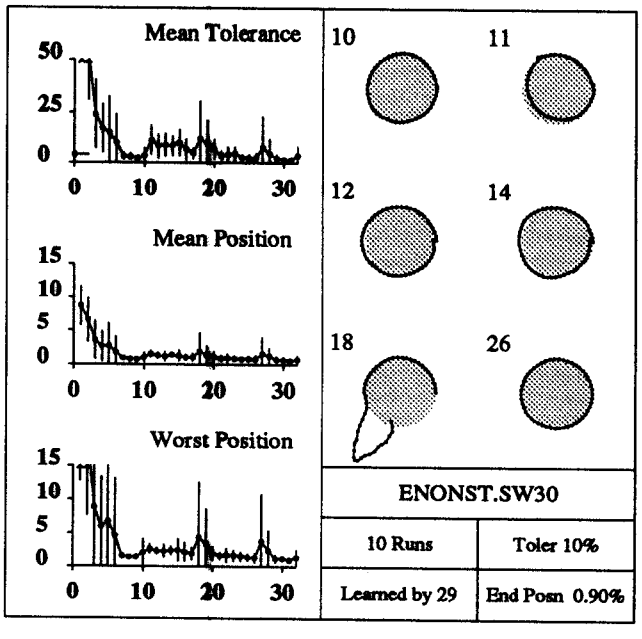

Figure 10.29

A non-stationary environment. The arm dynamics change on trial 10 to produce 1.5 times the torque that they used to. This experiment has a reasonable search width (6%), meaning that some old inaccurate exemplars are killed.

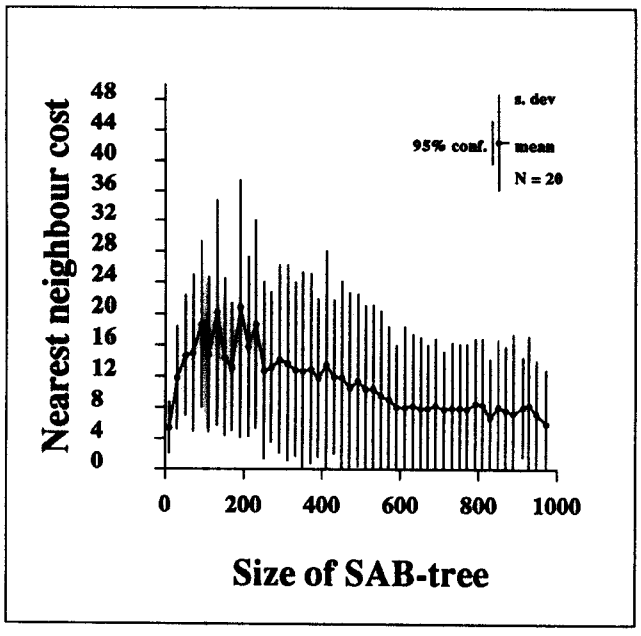

Figure 10.30

The number of *k*d-tree nodes processed. These results use binned averages from 20 independent trials.

The graph of the cost of search against tree size is shown in Figure 10.30. The result is remarkable: although the cost increases initially with tree size, it actually *decreases* as the tree gets larger! The typical number of distance calculations in the final trials is 6.

This is bizarre, but can be explained by the improving behaviour of the arm as learning progresses. The BASIC experiment achieved tolerable behaviour by, on average, the ninth trial. At this point, when partial inversion is used to obtain a candidate action, the requested behaviour and state are very close to the (correct) state-behaviour pair experienced on the preceding, also successful trial. Thus there is often a very nearly exact match for the nearest neighbour search to find, which usually means very little search backtracking is required.

To test this theory, we view the corresponding graph (Figure 10.31) for an experiment in which satisfactory performance was never attained. Here we use an experiment with a large amount of simulated noise. In this case it can be seen that the typical number of distance calculations is 15.

The conclusion is that, as hoped, there is empirical evidence that the nearest neighbour computation can be performed in real time because the distribution of exemplars is conducive to nearest neighbour search, and can actually *improve* the speed as the task is learned more accurately.

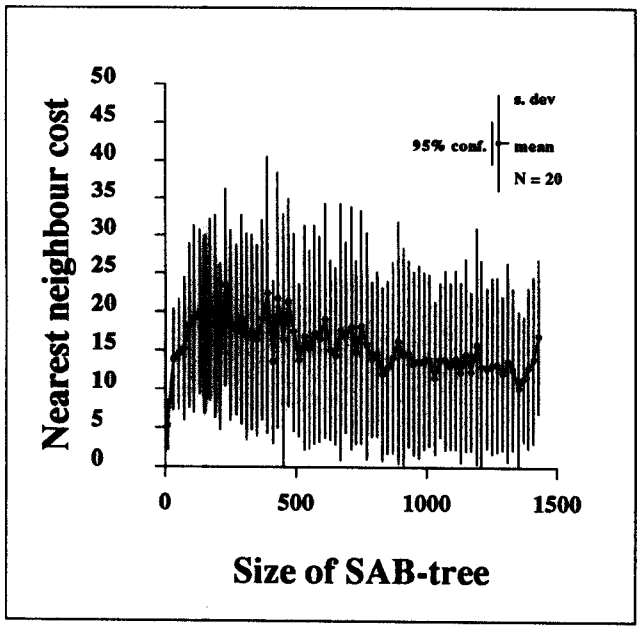

Figure 10.31

The number of *k*d-tree nodes processed. These results use binned averages from 20 independent trials.

10.4 Further Arm Experiments

These experiments are concerned with how the arm would perform if it were given a more complex task than tracking a prespecified trajectory. Many robot tasks, such as "pick and place" require that the robot move to a variety of positions, and it is interesting and important to see how performance is affected as the task demands that increasing subsets of the control space are learned.

The experiments also demonstrate the use of ice puck control, described in section 9.2. In these experiments trajectories are generated dynamically and autonomously. Figure 10.32 shows an example of a movement task, including the (stationary) start and goal states. The goal is to make the movement in 25 time steps. Figure 10.33 shows the ice puck trajectory generated to achieve this puck goal. Because puck trajectories are so cheap to compute they are recalculated every control cycle instead of being tracked. Figure 10.34 shows the trajectory recomputed after three very unskilled action choices had taken the hand far from the original trajectory.

Figures 10.35—10.38 demonstrate by means of animations how performance improves upon repetitive attempts at the movement task. It is seen to achieve the puck goal on only the third attempt, and after ten attempts the hand deviates very little from the straight line trajectory computed for the first step. This is indicated in Figure 10.39 which shows the ice puck trajectory generated three steps into a later trial. It matches almost exactly the initial generated trajectory of Figure 10.33.

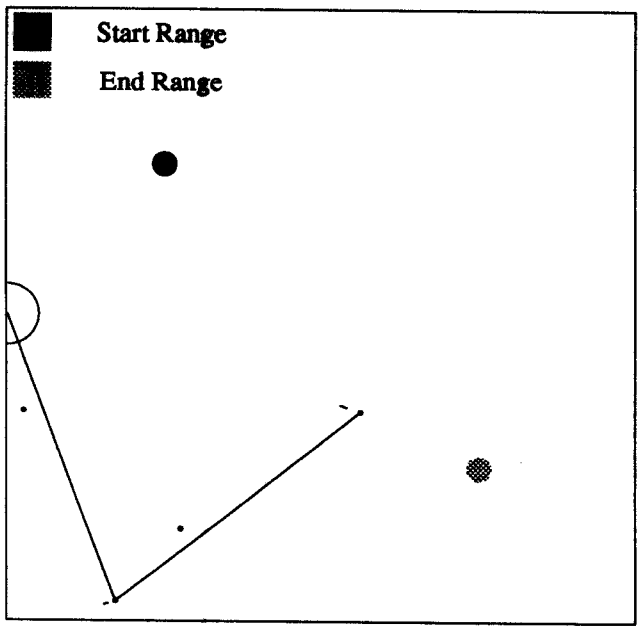

Figure 10.32

The start and end positions for the 1-d explore task. The start state and goal state both have zero velocity.

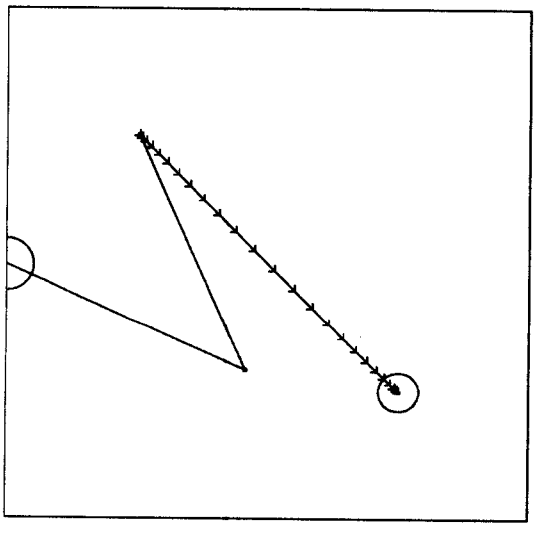

Figure 10.33: The ice puck trajectory to the goal state in 25 time steps, beginning at the exploration start state

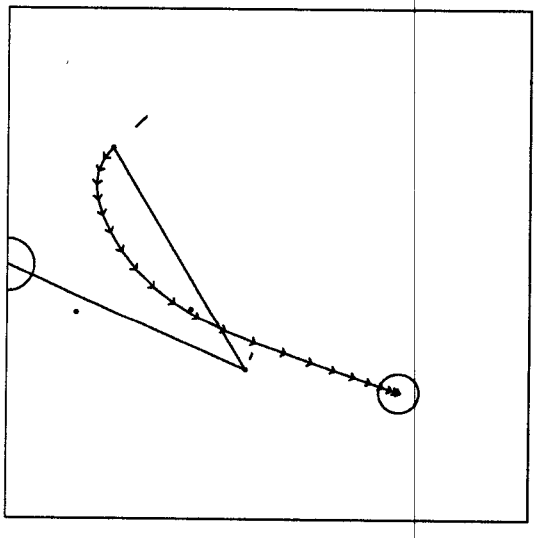

Figure 10.34: The ice puck trajectory to the goal state in 22 time steps, beginning at the state which the arm has arrived in after three, very unskilled, time steps.

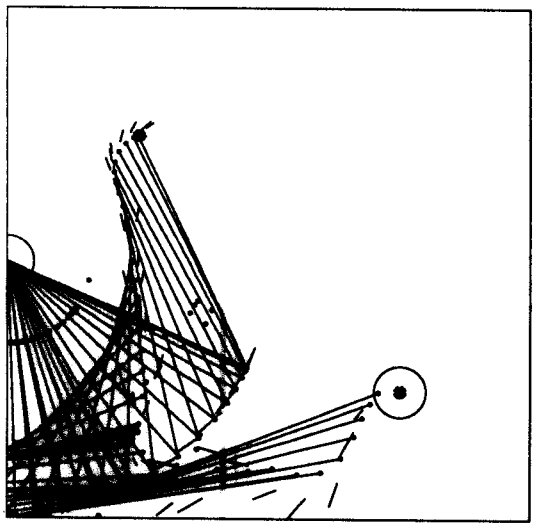

Figure 10.35: Animation of the first explore attempt. The goal time was 40 time steps, but this took much longer to even reach the target's neighbourhood.

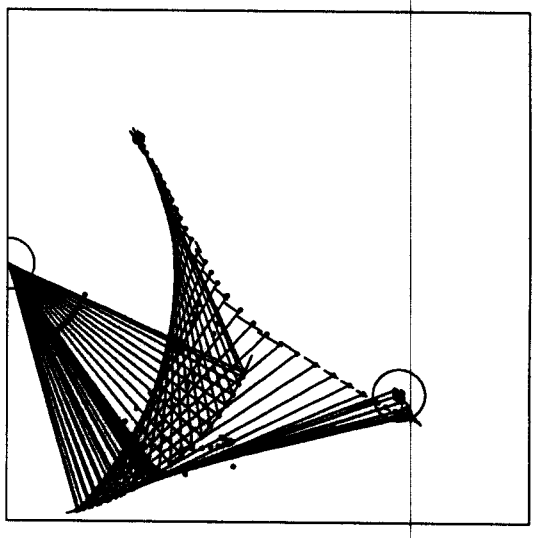

Figure 10.36: Animation of the 3rd explore attempt

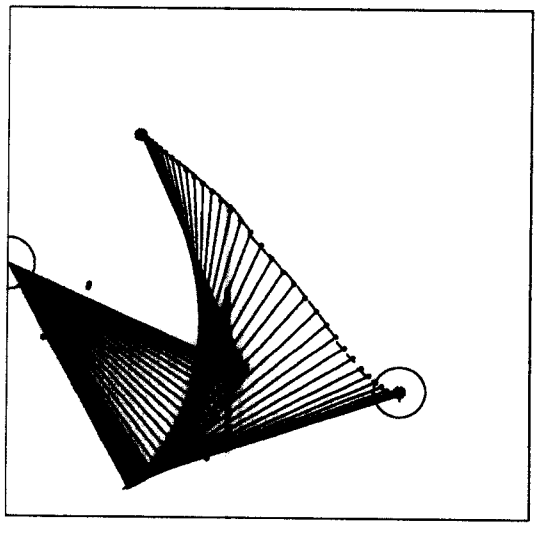

Figure 10.37: Animation of the 6th explore attempt

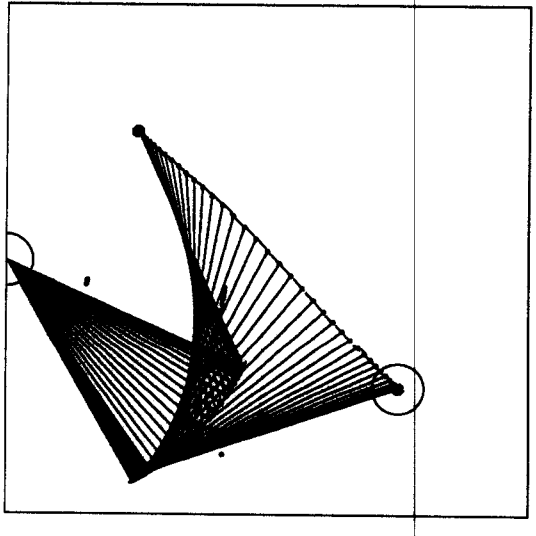

Figure 10.38: Animation of the 15th explore attempt

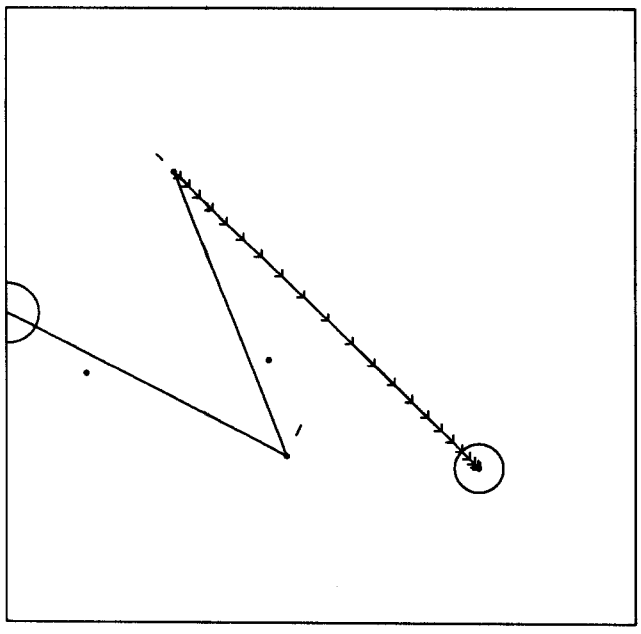

Figure 10.39

The ice puck trajectory to the goal state in 22 time steps, beginning at the state which the arm has arrived in after three well executed time steps. Notice that is nearly identical to the trajectory of Figure 10.33.

A more quantitative indication of the performance of the movement task is gained in Figure 10.40 which graphs the mean tolerance error for a number of movement task runs and Figure 10.41 which graphs the final position error. The final position error is a representative statistic of how well the task was performed. The puck goal was to achieve the position in forty time steps. The error after forty time steps was used as the statistic, but the trial was allowed to continue to try getting closer to the puck goal for a further period of time on the early occasions in which the hand did not achieve 3% position error or less within forty steps. The system parameters were all default.

10.4.1 Higher Dimensional Tasks

The previous movement task, once learned, was no harder than the trajectory tracking task, because each subsequent trial would require the same journey through state space. The task requires a one-dimensional strand through state space and so let us call it a *one-dimensional task*. To test how the performance would degrade with increasing dimensionality I devised some two, three, four and five-dimensional tasks. In each case the task generates a random start position and a random end position, each distributed uniformly within certain subspaces of the arm's reaching domain. Varying selections for the start and end distributions produce different dimensionality tasks. They are depicted in Figures 10.42—10.45.

An interesting thing to examine is the distribution of exemplars after the tasks of increasing dimensionality. Figures 10.46—10.49 show a snapshot of the position components

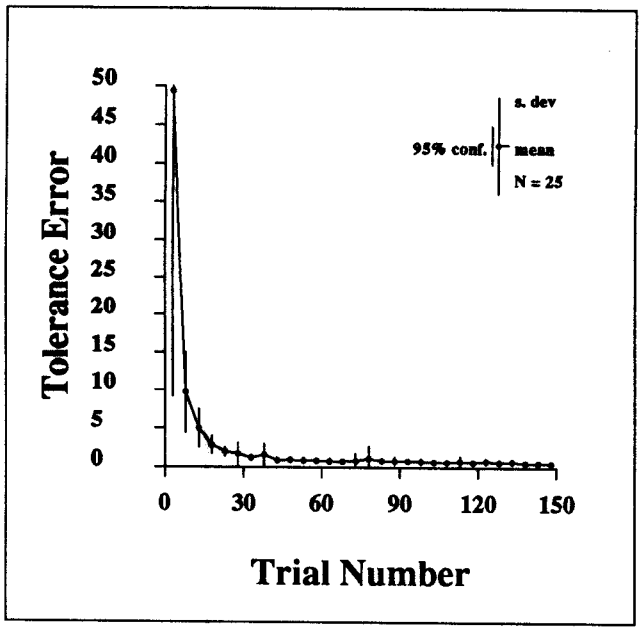

Figure 10.40

1-d exploration. The mean `Tolerance` error graphed against exploration attempt. This is the mean difference between acceleration requested by the Ice Puck controller and that obtained by the *SAB* action chooser. The range of possible accelerations is 300.

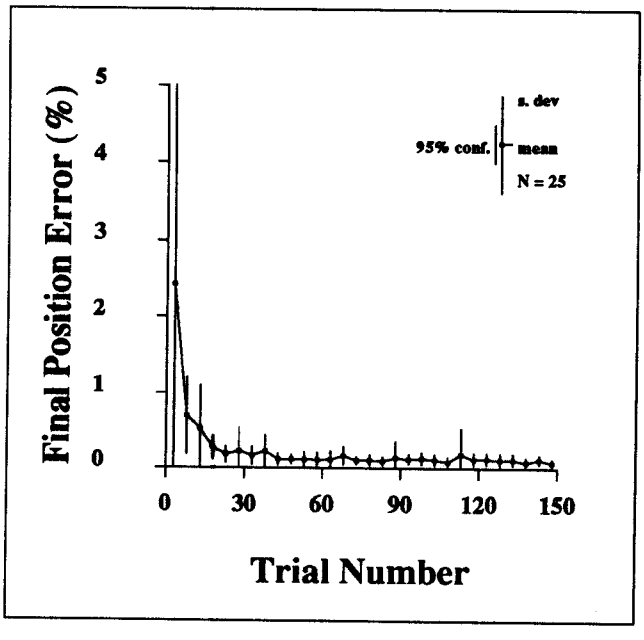

Figure 10.41

1-d exploration. The percentage final position error. This is the distance of the hand from the goal after 40 time steps, expressed as a percentage of the range of possible distances.

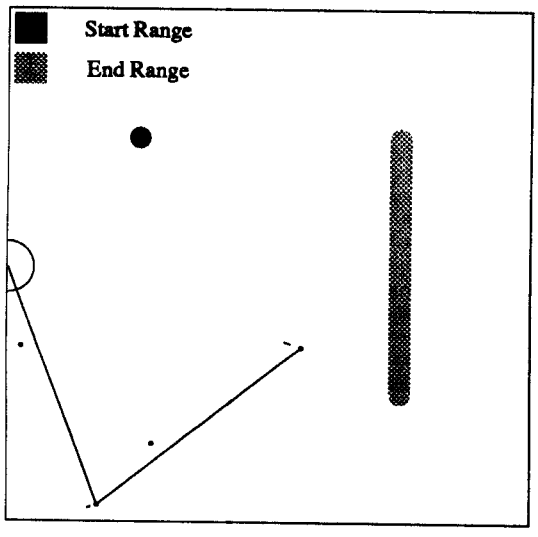

Figure 10.42: **2-d exploration.** The arm always starts in the same state, but the goal is chosen uniformly randomly from a 1-d set of states.

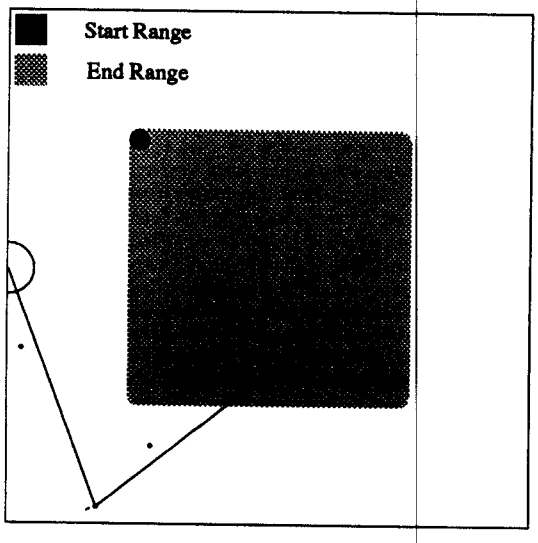

Figure 10.43: **3-d exploration.** The arm always starts in the same state, but the goal is chosen uniformly randomly from a 2-d set of states.

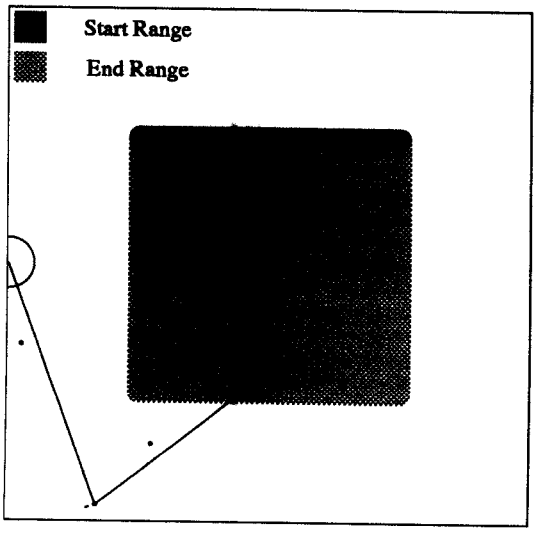

Figure 10.44: **4-d exploration.** The arm starts randomly, from a 1-d set of states. The goal is chosen uniformly randomly from a 2-d set of states.

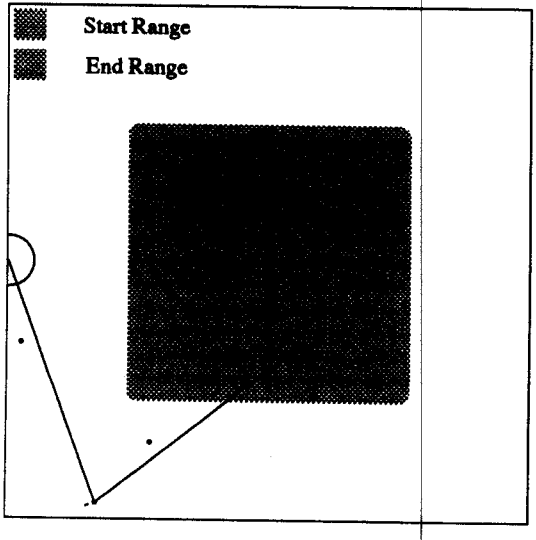

Figure 10.45: **5-d exploration.** The arm starts randomly, from a 2-d set of states. The goal is chosen uniformly randomly from a 2-d set of states.

Explore Dimension	Learned By	End Position (%)
1	8.3	0.11
2	9.3	0.2
3	12.4	0.36
4	20.9	0.5
5	26.9	0.39
5 (decomp.)	24.8	0.29

Table 10.17: Exploration performance summarized

of the exemplars in the *SAB*-tree after twenty random trials. They give an indication of how the dimensionality, and hence the amount that needs to be learned, increases. For example, the trajectories in the two-dimensional diagram all have similar speeds about half way through, which helps nearest neighbour to generalize better than in the three-dimensional tasks, in which the speed of the hand midway through trials varies more greatly.

The tolerance error graphs for these tasks are shown in Figures 10.50—10.53 and the final position error graphs in Figures 10.54—10.57. The graph data in summarized in Table 10.17, in which the "Learned by" statistic in the number of trials until the mean tolerance error was permanently less than $10ms^{-2}$, which corresponds to 0.3% of the range of possible tolerance errors.

The table indicates that learning is impaired by tasks of higher dimensionality. However, the nearest neighbour generalization is in this case sufficiently powerful that performance, in each case, does eventually reach an adequate level. This behaviour is not guaranteed in general, and so one solution might be, when a high dimensional task is required, to use an abstract controller which breaks it up into low dimensional tasks. This is exemplified by the five-dimensional *decomposed* task. Instead of moving from a random start position to a random end position in forty time steps, the controller uses two tasks:

1. In twenty time steps, take the hand from the random start position to near a stationary state at the centre of the space.

2. Then, in a further twenty time steps move to the random goal position.

The task description, exemplar distribution, tolerance graph and position graph are shown in Figures 10.58—10.61. The summary statistics are included in Table 10.17. It is seen that both learning speed and accuracy improve as a result. It is conjectured that for even higher dimensional tasks in higher dimensional state spaces this kind of dimension reduction at the control level will rather than merely improving performance, be essential for a reasonable learning rate.

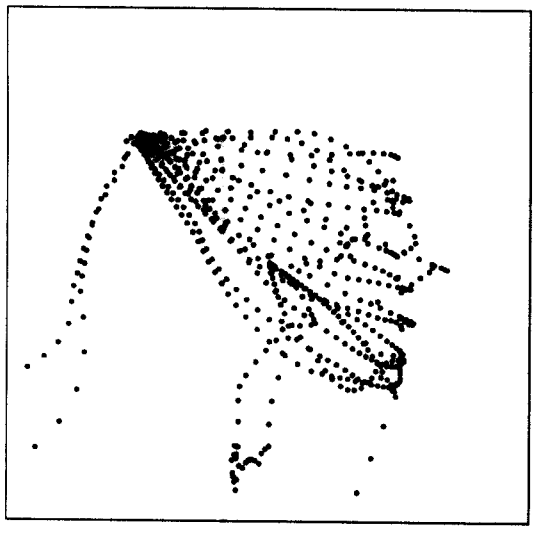

Figure 10.46: **2-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

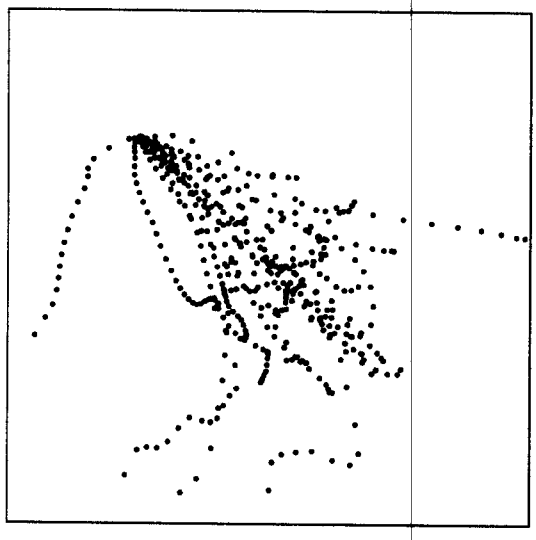

Figure 10.47: **3-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

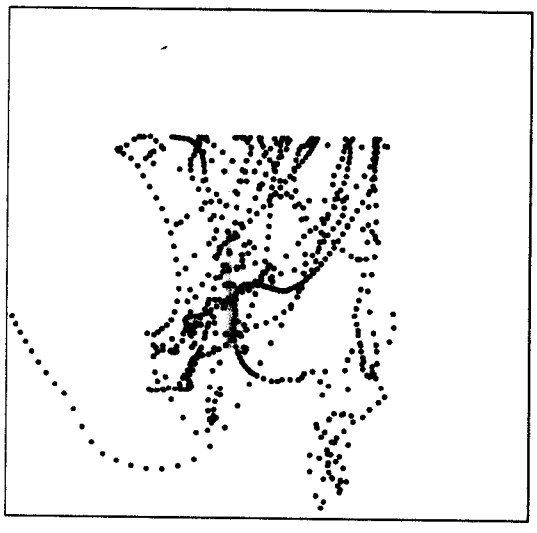

Figure 10.48: **4-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

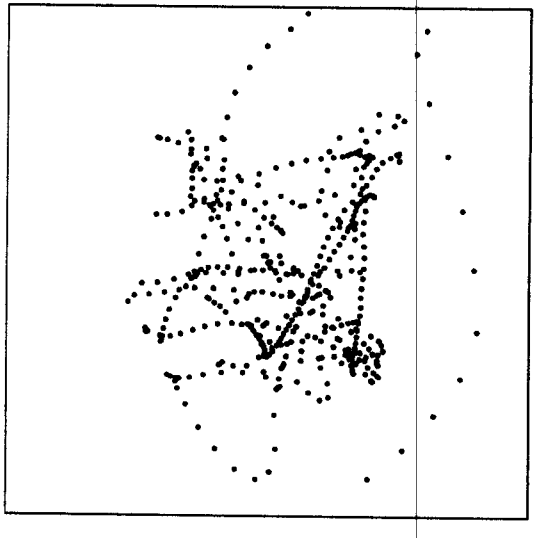

Figure 10.49: **5-d exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

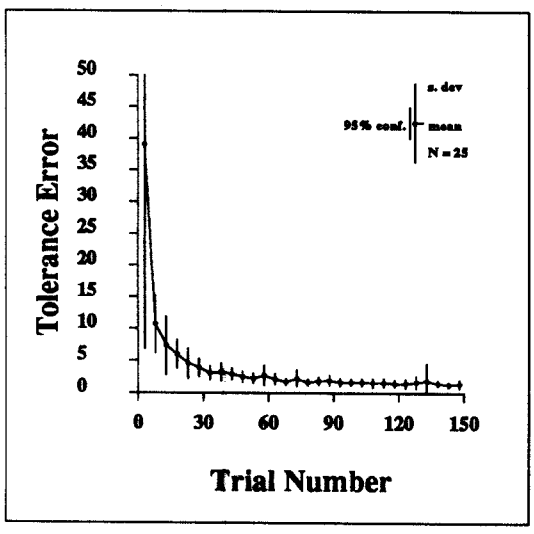

Figure 10.50: **2-d exploration.** The mean `Tolerance` error.

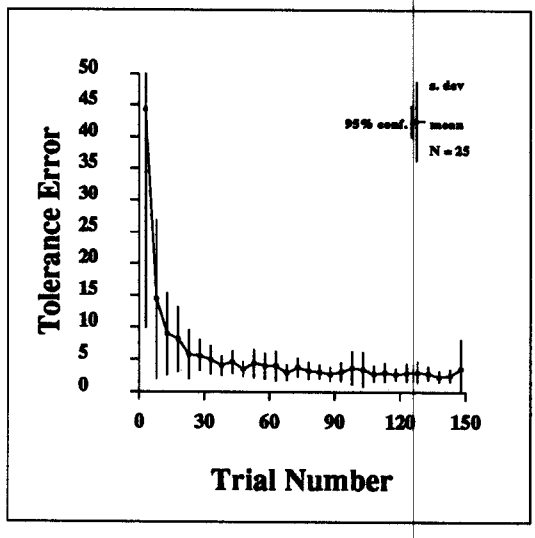

Figure 10.51: **3-d exploration.** The mean `Tolerance` error.

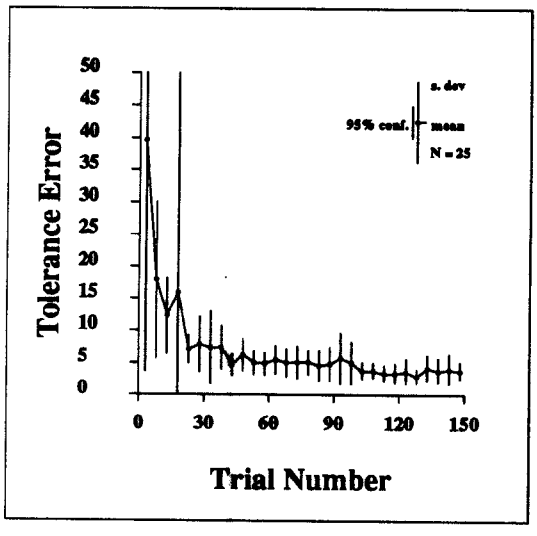

Figure 10.52: **4-d exploration.** The mean `Tolerance` error.

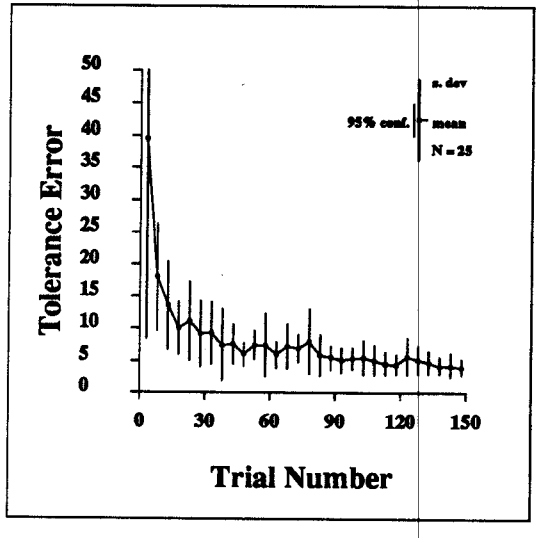

Figure 10.53: **5-d exploration.** The mean `Tolerance` error.

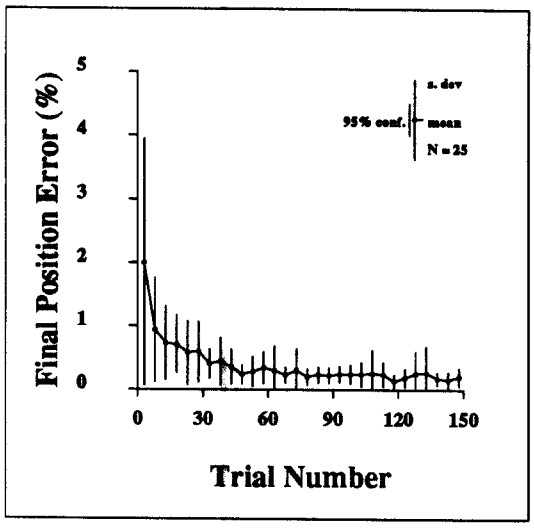

Figure 10.54: **2-d exploration.** The mean final position error, expressed as a percentage.

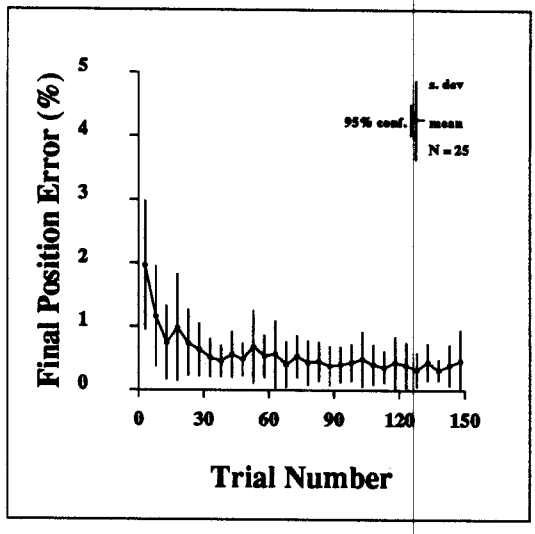

Figure 10.55: **3-d exploration.** The mean final position error, expressed as a percentage.

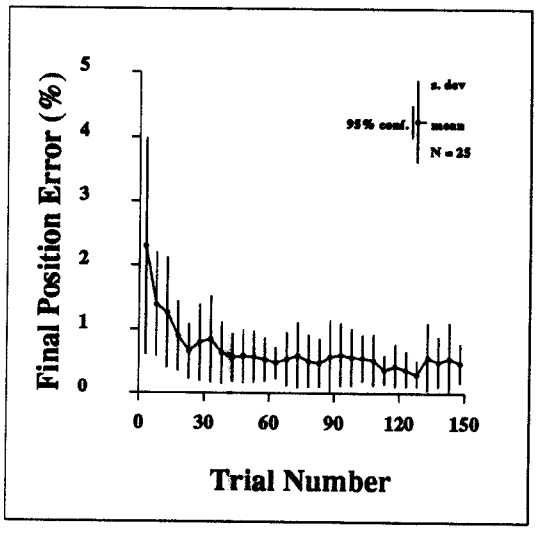

Figure 10.56: **4-d exploration.** The mean final position error, expressed as a percentage.

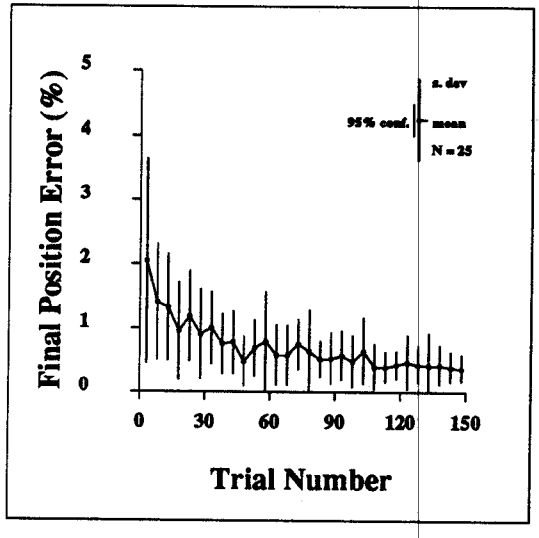

Figure 10.57: **5-d exploration.** The mean final position error, expressed as a percentage.

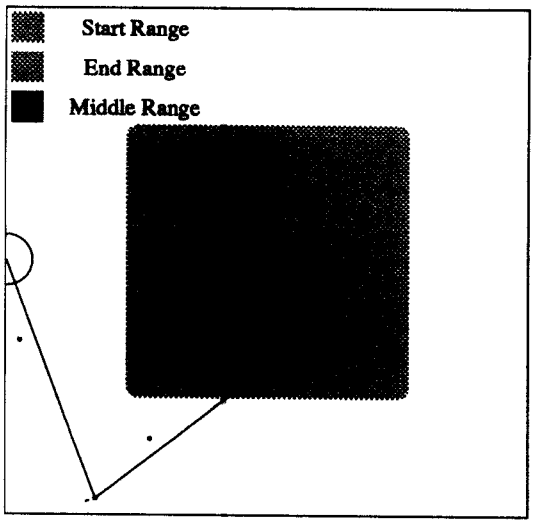

Figure 10.58: **5-d decomposed exploration.** The arm starts randomly, from a 2-d set of states. The goal is chosen uniformly randomly from a 2-d set of states. The controller splits up the problem to reach the stationary middle state half way through path.

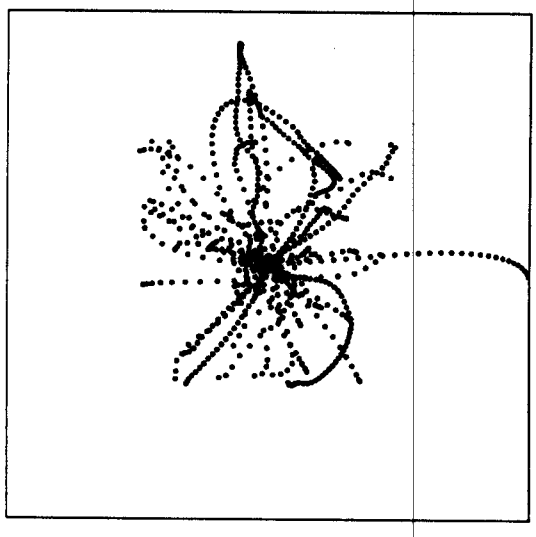

Figure 10.59: **5-d decomposed exploration.** Distribution of position components of exemplars in the *SAB*-tree after 20 trials.

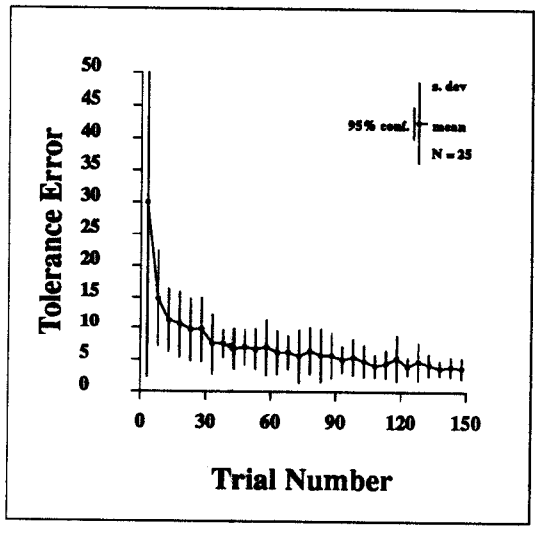

Figure 10.60: **5-d decomposed exploration.** The mean `Tolerance` error.

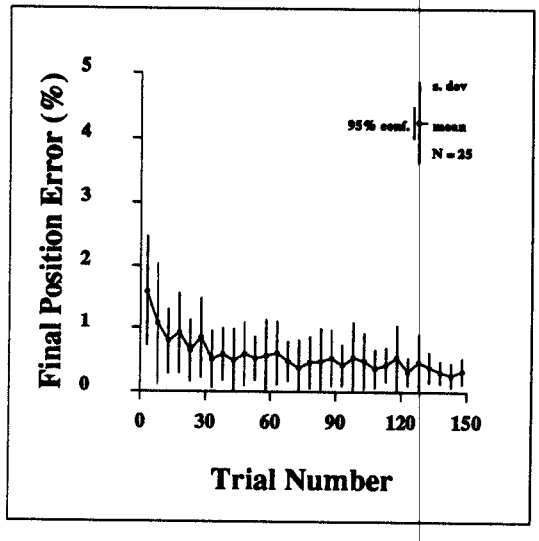

Figure 10.61: **5-d decomposed exploration.** The mean final position error, expressed as a percentage.

10.4.2 Autonomous Time Choice

In the experiments of the previous section the goal was a specified position in forty time steps. Here, we make the task more abstract and simply specify the goal position and leave it up to the world model-based controller to arrive at the goal in a reasonable time, which varies according to how far apart the start and end states are, and where they are situated. The abstract controller here is used as an indication of the kind of simple mechanism which can be used, based on learned world models, to increase the abstraction of the task. It is not to be considered as a serious attempt at a robust optimal controller.

The decision of how many time steps are required is achieved by means of the strength estimation mechanism of Section 8.3. The controller is told the start and end states for the task, and then performs a binary search of possible times to goal. For each possible time to goal it estimates if it has the strength to achieve the time to goal, using an ice puck trajectory. It does this by generating the ice puck trajectory, and then using the strength mechanism to decide if

1. With probability greater than P_{res} (specified by system parameter **presig**, and given value 0.2), twice the required acceleration at the start state can be obtained.

2. With probability greater than P_{res} twice the required acceleration half way through the trajectory can be obtained.

3. With probability greater than P_{res} twice the required deceleration at the end of the trajectory can be obtained.

This is only a heuristic, and is certainly not likely to be near optimal control. Twice the required acceleration is used in the strength estimation to provide a degree of robustness.

The results for autonomous time choice are shown in Tables 10.18 and 10.19. The times to achieve the goal are all, on average, quicker than forty steps used in the previous task. Learning is slower in all cases because (i) the increased average speed means an increased diversity of the accelerations requested of the SAB action chooser and (ii) some extra variation is introduced in the tasks in the initial stages before accurate estimates of strength become available. By the standard of the "Learned by" criterion, the 5-d task does not learn within 150 trials, although the 5-d decomposed task does. The decomposed task is necessarily slower, because the indirect route to the goal is often less efficient.

10.4.3 Disordered Environment

The autonomous-time-choosing 5-d trial was run in a noisy environment. The noise level was sufficient to randomly corrupt the acceleration by typically up to 5ms^{-2} which is 1.5% of the range of accelerations. Whilst subjected to this uncontrolled noise, both tolerance error and final position error are seen to improve with trial number in Figures 10.62 and 10.63.

Explore Dimension	Learned By	End Position (%)
1	18.6	0.36
2	31	0.43
3	43.5	0.46
4	87.9	0.44
5	Didn't Learn	0.5
5 (decomp.)	64.1	0.57

Table 10.18: Autonomous time-choice exploration performance summarized

Explore Dimension	Mean time steps to goal
1	24
2	22
3	24
4	29
5	28
5 (decomp.)	36

Table 10.19: Autonomous time-choice: mean time per trial

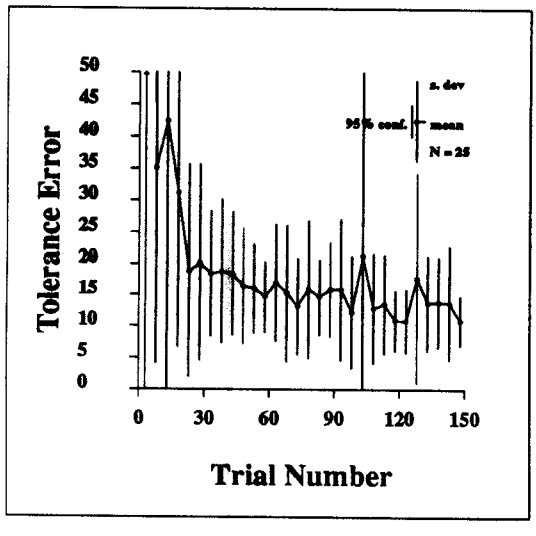

Figure 10.62: 5-d exploration in a noisy environment. The mean Tolerance error.

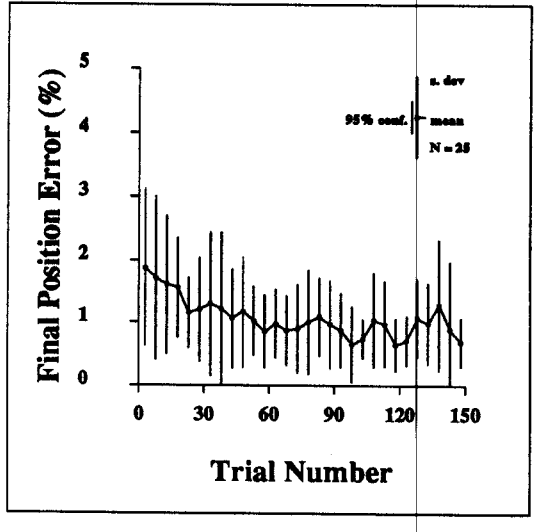

Figure 10.63: 5-d exploration in a noisy environment. The mean final position error, expressed as a percentage.

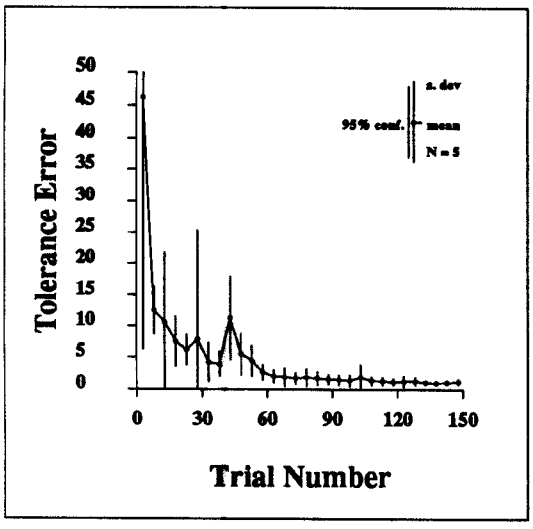

Figure 10.64: 1-d exploration in which the task changes on trial 40. The mean `Tolerance` error.

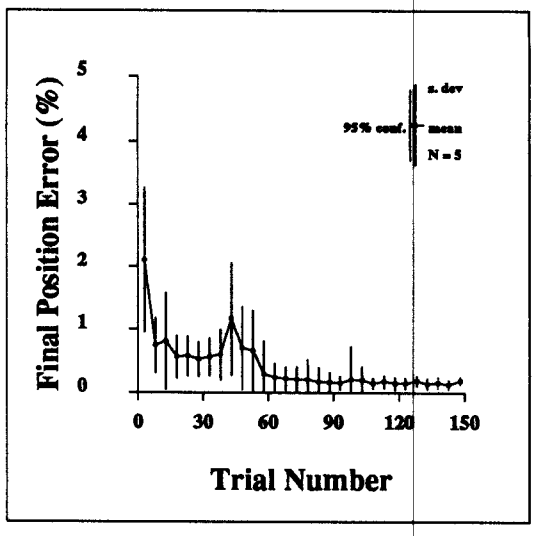

Figure 10.65: 1-d exploration in which the task changes on trial 40. The mean final position error, expressed as a percentage.

The next experiment was to confirm a feature of *SAB* learning which was confidently predicted prior to the experiment—that concentration on one particular task cannot cause "over-learning". An autonomous-time-choosing 1-d trial was run for forty trials using the task in Figure 10.32. The task was then changed to an entirely different start and end position for the remaining 110 trials. Figures 10.64 and 10.65 show the results.

The final experiment was adaptation to a changing environment for the 5-d autonomous-time-choosing task. For this experiment the `Search-width` parameter was tripled. It can be seen in Figures 10.66 and 10.67 that neither improvement nor reduction in performance occurred, except to a small extent for the Tolerance error, which tended to increase slightly for approximately thirty trials after the change. The reason for the lack of large effect is likely to be because the distribution of exemplars was so wide (due to the task's high dimensionality, compounded by variations due to autonomous time choice) that misleading exemplars were not noticeably inaccurate, and were possibly more use than no information at all.

10.5 Juggling a Ball

This simulated experiment is motivated by a similar (but real) experiment by [Aboaf *et al.*, 1989] A visually observed ball is bounced on a two-dimensional surface (see Figure 10.68). The flat surface can be moved upwards at a specified speed and angle. The following information is observed once every bounce:

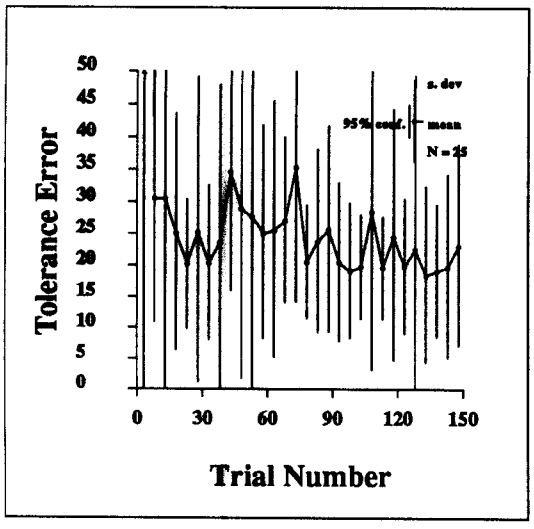

Figure 10.66: 5-d exploration in a non-stationary environment. The mean Tolerance error. Dynamics change on trial 40.

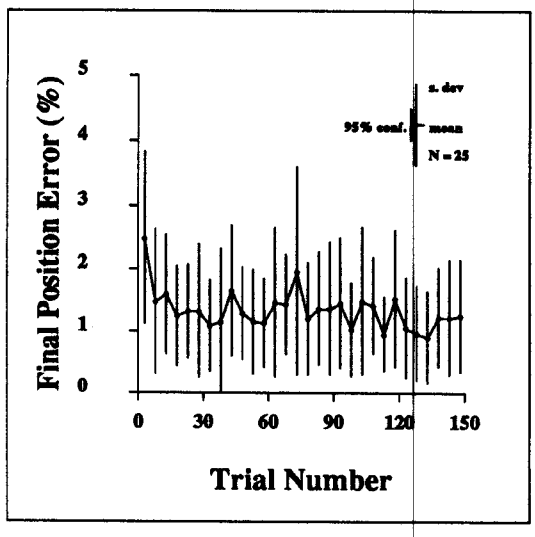

Figure 10.67: 5-d exploration in a non-stationary environment. The mean final position error, expressed as a percentage. Dynamics change on trial 40.

- The x coordinate of the ball, x_{top}, when the ball reaches the top of the bounce.
- The y coordinate of the ball, y_{top}, when the ball reaches the top of the bounce.
- The horizontal speed of the ball, u_{top}, when the ball reaches the top of the bounce.

The vertical speed is zero at the top of a bounce. The three values $(x_{\text{top}}, y_{\text{top}}, u_{\text{top}})$ constitute the perceived state of the system.

The ball is controlled by the surface. The surface is always reset to the same height before the ball reaches the top of the bounce. Then, a short period of time, t_{go}, after the ball reaches the top of its trajectory, the surface centre starts to move upwards with constant speed v_{go}. The bat is also placed at angle θ_{go} to the horizontal. Thus the three variables t_{go}, v_{go} and θ_{go} affect the behaviour of the surface and constitute the action used to control the ball. The permissible values of the state and action components are shown in Table 10.20. If any state variable exceeds its range the ball is deemed to have crashed. The following PSTF was learned:

$$\underbrace{(x_{\text{top}}, y_{\text{top}}, u_{\text{top}})}_{\textbf{State}} \times \underbrace{(t_{\text{go}}, v_{\text{go}}, \theta_{\text{go}})}_{\textbf{Action}} \rightarrow \underbrace{(x'_{\text{top}}, y'_{\text{top}}, u'_{\text{top}})}_{\textbf{Behaviour}} \tag{10.3}$$

where $(x'_{\text{top}}, y'_{\text{top}}, u'_{\text{top}})$ is the perceived state of the subsequent bounce.

The abstract controller was based on the notion of an ideal state: $x_{\text{top}} = 0, y_{\text{top}} = 1, u_{\text{top}} = 0$. On each control cycle the controller attempts to apply an action which at least halves the current error of each state variable. The error of a state variable

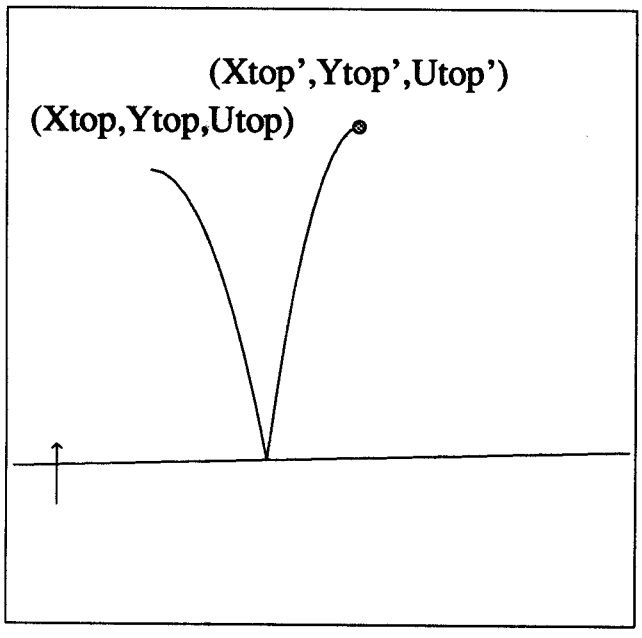

Figure 10.68

A single ball bounce. The surface was moving upwards, and was at a slight angle.

-1.5	$\leq$	x_{top}	$\leq$	1.5
0.5	$\leq$	y_{top}	$\leq$	2
-1	$\leq$	u_{top}	$\leq$	1
0	$\leq$	t_{go}	$\leq$	0.5
-0.5	$\leq$	v_{go}	$\leq$	0.5
-0.2	$\leq$	θ_{go}	$\leq$	0.2

Table 10.20: Ball bouncing: permissible state and action values.

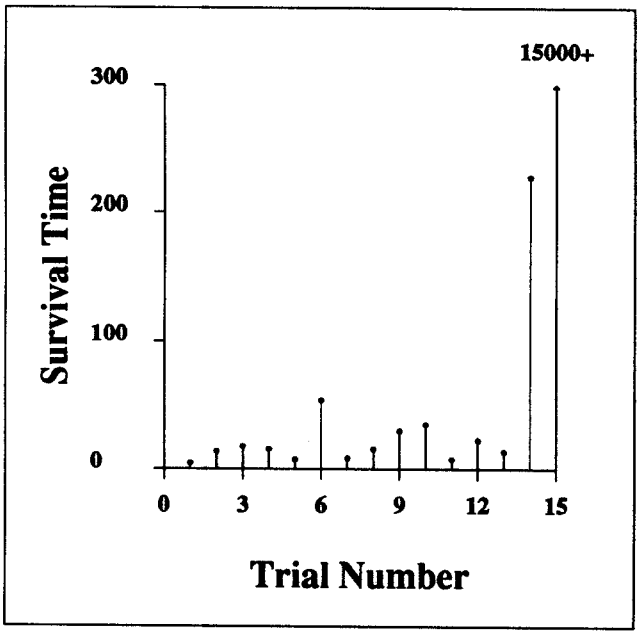

Figure 10.69

The number of bounces before disaster against trial number. Trial 15 was halted, still bouncing at 15000.

is its absolute difference from its ideal value. For example, if the current state were $x_{\text{top}} = 1, y_{\text{top}} = 0.8, u_{\text{top}} = -0.2$ then the *SAB* action chooser would be requested to find an action to produce a next state for which

$$\begin{aligned} -0.5 &\leq x_{\text{top}} \leq 0.5 \\ 0.9 &\leq y_{\text{top}} \leq 1.1 \\ -0.1 &\leq u_{\text{top}} \leq 0.1 \end{aligned} \tag{10.4}$$

This abstract controller is more robust than one which aims precisely for the ideal state—if the ball is in danger of crashing it is of primary importance to prevent its state getting worse and only of secondary importance to try to send it immediately to the ideal state. It should be noted that this abstract controller is not trivial in design, and it required some of the human designer's qualitative domain knowledge.

10.5.1 Results

The graph of Figure 10.69 shows the number of bounces against trial number. The simulated world has 2% noise in each variable and all the *SAB* system parameters were default.

The graph shows that for the first 13 trials the number of bounces before disaster was usually between three and thirty. Trial 14 was successful for over 200 bounces, but there was an eventual serious mistake. Trial 15 was halted after four hours of simulation had produced over 15,000 successful bounces. Figure 10.70 shows the behaviour of the x_{top}

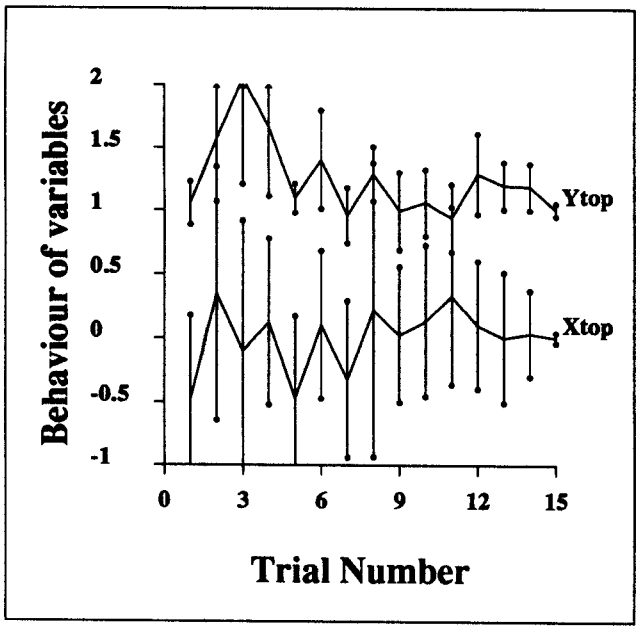

Figure 10.70

The mean and standard deviation of the x_{top} and y_{top} coordinates experienced during the juggle trial.

and y_{top} components of state. They varied greatly until trial 15, when they were kept safely within 5% of the centre of their ranges.

Further experiments were performed with 5% noise in the perceived state observations. Figures 10.71 and 10.72 show performance respectively without and with the *SAB* smoothing mechanism. In the second case the range width D_{range} was 5% of the width of the *SAB* tree domain. The results provide evidence that smoothing the observations improves performance. It should be noted that despite the inferior results of the unsmoothed experiment, its juggling behaviour was still good, with several trials of more than 1000 bounces. The smoothed experiment was run three other times, on each occasion halted after 10,000 bounces. The successful trial numbers were respectively 16, 20 and 20. A smoothing kernel of twice the width (10% of the domain width) was also tried. It was halted after 5,000 bounces on trial 21. The large smoothing width caused an approximate five-fold decrease in computation speed.

Graphs 10.73 and 10.74 show the performance in a changing environment. The ball's coefficient of elasticity increases during trials (it starts at 0.8 and increases to 1 asymptotically). In Figure 10.73 the initial change is sufficiently quick that by trial 70 the *c.o.e.* is very close to 1, and so no further apparent change occurs. Only after changes have become negligible is there a trial that avoided disaster, indicating that the adaptation mechanism was not able to help *during* the environmental change. In Figure 10.74 the changes are significant until approximately trial 180. There were no disaster-free trials before the experiment was halted on trial 220.

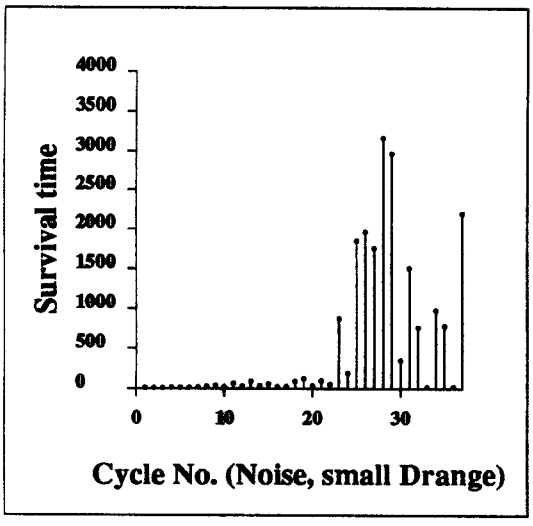

Figure 10.71: The number of bounces before disaster against trial number in a noisy environment: noise 5% and no smoothing.

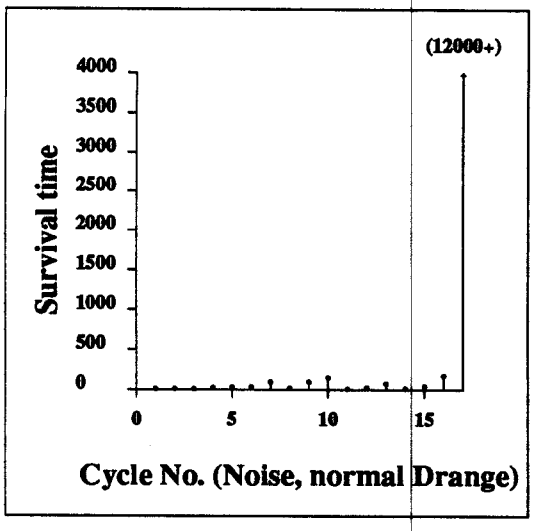

Figure 10.72: The number of bounces before disaster against trial number in a noisy environment: noise 5% and using a smoothing kernel of width 5%.

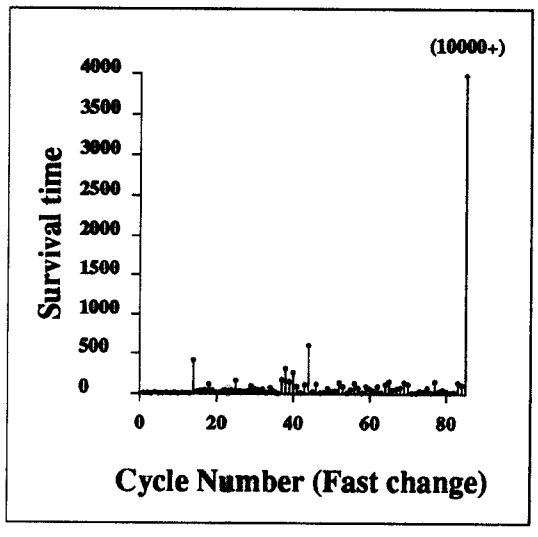

Figure 10.73: The number of bounces before disaster against trial number in a changing environment. Changes have become negligible by trial 70.

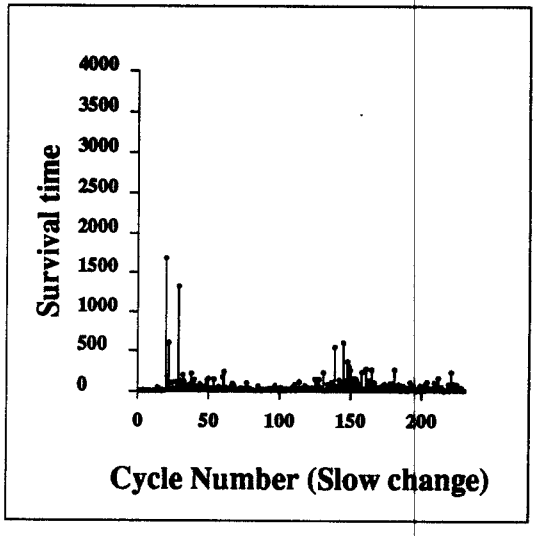

Figure 10.74: The number of bounces before disaster against trial number in a slowly changing environment. Changes are negligible by trial 180.

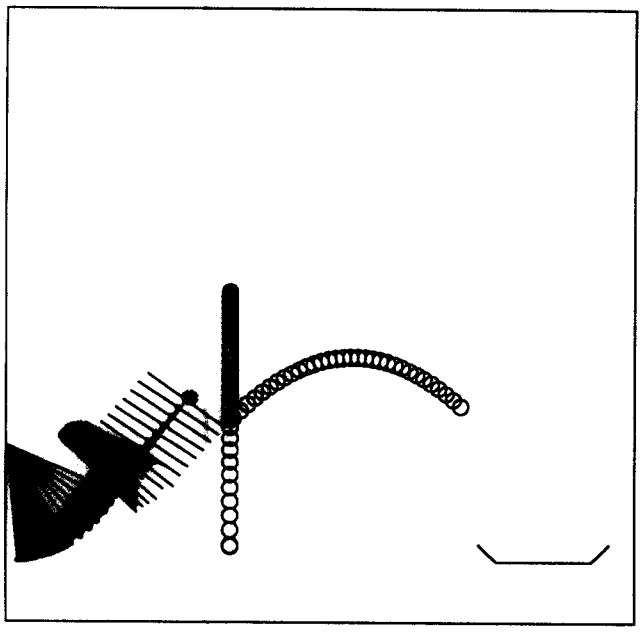

Figure 10.75

An early volley attempt. The ball is fired from the right towards the arm. From the initial state of the ball the controller erroneously predicts it will arrive at the grey circle. The ball is clipped by the bat and flies up vertically before falling to earth.

10.6 The Volley Task

The volley task was described in Section 9.5. It is an example of a compound task which has several learning components and is considerably larger and more complex than typical learning control tasks described in the literature. However, it *is* simulated. There is significant simulated noise to capture the same sort of disorder that a real experiment would have. The task is decomposed into a hierarchy of learning controllers which involve three *SAB*-trees which learn and control the following:

- The perceived state transition function of the arm.
- The dynamics of the bat/ball collision.
- The behaviour of the ball (used for prediction only).

The new *SAB*-trees have default parameters except for the scaling of the components. The components were not difficult to scale—they are all either speeds and positions in the simulated world, and so used the same scaling as that used for the arm.

The simulated ball has noise in its dynamics: enough to vary the arrival position by up to approximately 3% of the length of the arm.

Figures 10.75 and 10.76 show the behaviour of the arm during an early and late trial respectively.

Figure 10.77 graphs the performance of a relatively easy task, where the ball is always fired from the same position and velocity and the bucket always remains in the same place.

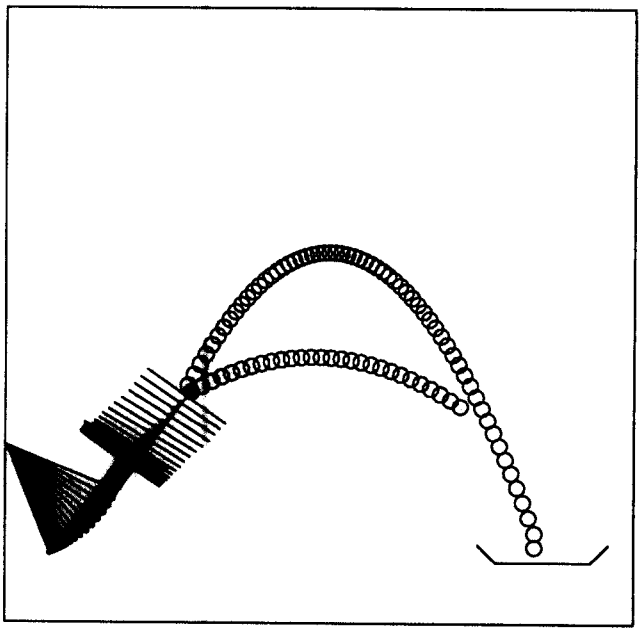

Figure 10.76

A successful volley. As well as modelling its own arm, the controller has correctly predicted where the ball will be when it is in range and found a correct speed with which to hit the ball back to the bucket.

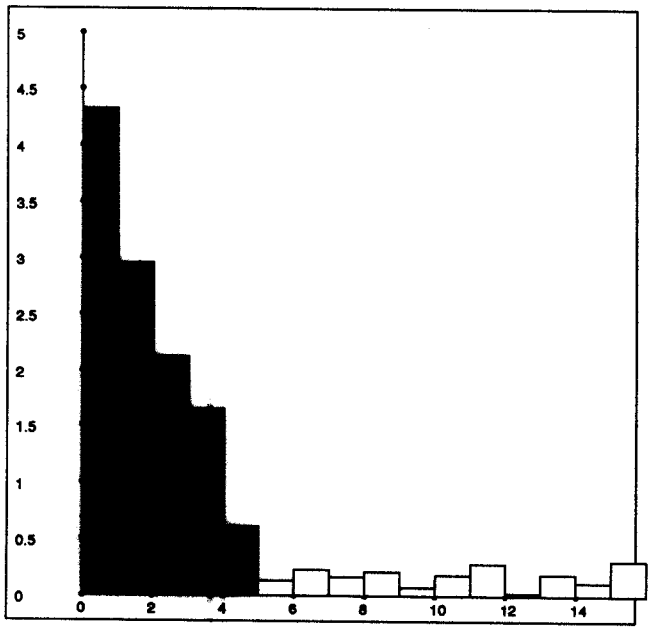

Figure 10.77

A histogram of distance from bucket against trial number. The successful volleys, which landed in the bucket, are shown in white. During these trials the bucket was fixed and the ball always fired with the same speed and direction.

After five trials a suitable hitting speed is discovered, and a value close to this speed is used for subsequent trials.

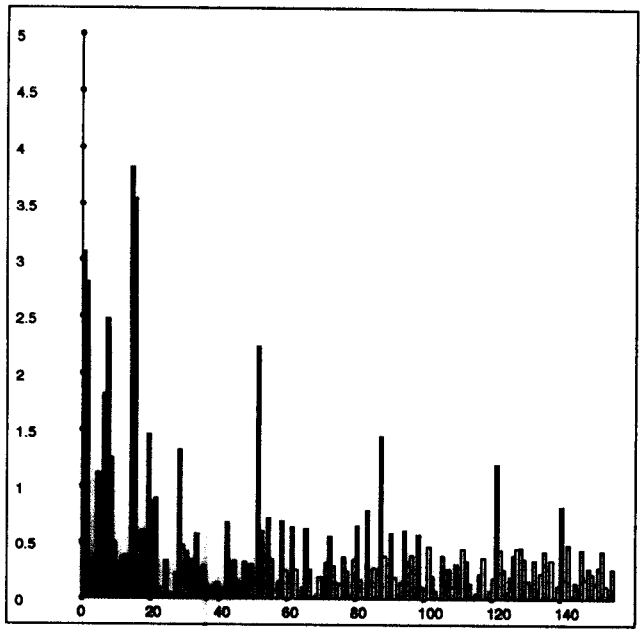

Figure 10.78

A histogram of distance from bucket against trial number. Before each trial the bucket was placed at a random position. The ball was always fired with random speed but constant direction.

Figure 10.78 displays the results of the considerably harder task in which the ball is fired at a random speed for each trial, and the bucket is placed in a random position. It requires approximately twenty trials before the behaviour can be said to be fairly skilled.

Figure 10.79 shows the behaviour when the bucket is placed randomly and the ball is fired with a random speed *and* direction. There is an improvement in behaviour but the probability of failure is still roughly 20% even after over 100 trials. This is because even then there is still a fair probability that the starting state of the ball is sufficiently far from any previous experience that the behaviour predicted by the nearest neighbour is inadequate.

Thus learning behaviour is demonstrated for a complex robotic system. The learning was efficient—approximately 30 seconds per volley attempt initially, rising to 3 minutes per attempt when the *SAB*-trees were large enough to cause substantial virtual memory swapping on the computer. The final size of the PSTF *SAB*-tree was approximately 50,000 exemplars.

10.7 Experimental Results: Conclusions

The experimental results have generally confirmed the goals of this investigation. The learning rate was very fast—only a small number of trials of any experiment were required. Indeed, the testing was (by the standards of much previous research for which learning speed was not top priority) somewhat severe in that an experiment was usually classed as

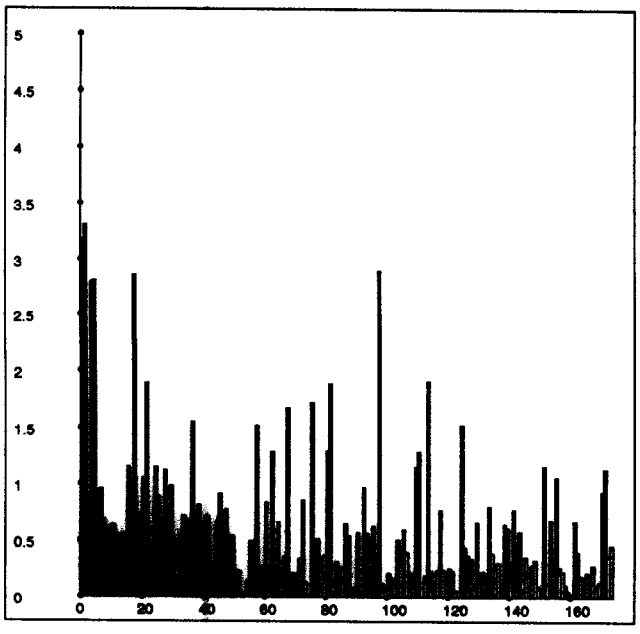

Figure 10.79

A histogram of distance from bucket against trial number. Before each trial the bucket was placed at a random position. The ball was always fired with random speed and random direction.

unsuccessul if learning had not occurred within a few dozen trials. It was also demonstrated that the rate of learning is not badly affected by increased accuracy requirements. In the "hold still" task of Section 10.2 the time to achieve any given tolerance was seen to increase only logarithmically with the reciprocal of the tolerance.

The learning was also computationally efficient, even with *k*d-trees of up to 10,000 exemplars. This was particularly noticeable for the hand-eye experiment in which the real time to learn, including waiting for arm movements and image processing was only in the order of thirty minutes. Learning did become noticeably slower when *k*d-trees were several tens of thousands of exemplars large. This is conjectured to be because of the extravagant use of memory in the current implementation of the *SAB*-trees, which requires 256 bytes per exemplar, causing a 20,000 node tree to use 4 mageabytes of store which causes substantial virtual memory swapping. Use of very large range search widths also slowed computation. This is because *SAB*-tree update then involved accessing a significant fraction of all exemplars.

The learning was robust. It was demonstrated that all system parameters could be varied considerably without significant detrimental effect on performance. Futhermore, no experiment became "stuck". The smoothing mechanism was demonstrated to help combat noise in the arm experiments and the juggle experiments.

The change adaptation mechanism was also shown to have some beneficial effect for the trajectory tracking and juggling tasks, but for the high dimensional arm task the results were disappointing. A worthwhile area for further study would be to improve

the pruning of old inaccurate exemplars by more sophisticated (and probably off-line) statistical analysis.

Finally, a significantly complex task—volleying—was controlled successfully by a hierarchy of *SAB* learners. Learning took place simultaneously at different layers of abstraction, demonstrating how learning world models can be of considerable help even for high level robotic tasks.

Chapter 11

Some Extensions

This chapter reviews some extra topics which were researched during the course of the dissertation, but which are not central to its main ideas. In Section 11.1 Albus' CMAC is discussed in further detail, in particular its relationship to exemplar-based methods. In Section 11.2 the use of Dynamic Programming in conjunction with SAB-learning is examined and extended.

11.1 Albus' CMAC and kd-trees

The Cerebellar Model Articulation Controller (CMAC) is a connectionist model, devised by J. S. Albus in the early 1970s, for learning smooth, continuous, non-linear mappings [Albus, 1975a; Albus, 1975b; Albus, 1981]. It is specifically useful where a high degree of global generalization is positively not wanted, but local generalization to take advantage of the assumed smoothness is welcome. In the next section I will summarize its implementation and behaviour. Although the implementation is connectionist, I then propose an alternative *k*d-tree implementation with exactly the same behaviour, called "Symbolic CMAC". I conclude by comparing the behaviour of CMAC with that of the nearest neighbour generalization.

11.1.1 CMAC Summary

CMAC implements a mapping from a quantized k_s-dimensional space $\mathbf{S}$ to a real-valued k_p-dimensional space $\mathbf{P}$. When an input vector is presented either for access or update it is preprocessed to obtain a number of indices into a large array of weights. These indices are chosen in a manner (detailed in [Albus, 1975a]) which causes vectors close to each other in S-space to have a large proportion of shared indices. The notion of closeness is defined by a metric $dist : \mathbf{S} \times \mathbf{S} \rightarrow \Re$.

The closer two vectors are, the more indices they share. Conversely, beyond a certain distance they share no common indices. Let us call the set of indices associated with an

input vector X the *neighbourhood* of X. The indices address array entries which contain k_p-dimensional weights.

When CMAC is being accessed these weights are all added to produce the predicted vector from **P**-space. Write $W_N(i)$ as the weight corresponding to the index i at learning iteration N. Let *nhood*(X) denote the set of indices associated with input vector X. The value predicted for input X on the Nth iteration is

$$I_N(X) = \sum_{i \in nhood(X)} W_N(i). \tag{11.1}$$

When CMAC is updated the update increment is distributed evenly among all the indices in the neighbourhood. If on the Nth learning iteration we wish to increment the predicted value at X_N by the value δ_N, the weights are adjusted thus:

$$W_{N+1}(i) = \begin{cases} W_N(i) + \frac{1}{C}\delta_N & \text{if } i \in nhood(X_N) \\ W_N(i) & \text{otherwise.} \end{cases} \tag{11.2}$$

Where $C = \mid nhood(X_N) \mid$. There seems no reason to vary the number of indices in different neighbourhoods, and so generally C is a system constant independent of the neighbourhood. Conventionally the initial value of all weights is zero so $I_N(X) = 0$ for all X.

When the learning data is presented as a set of (X_N, P_N) pairs, the increment on the Nth iteration is obtained by first finding the current predicted value $I_N(X_N)$, then evaluating the error $P_N - I_N(X_N)$ and incrementing CMAC at X_N by this error. An alternative increment, used by [Miller *et al.*, 1987; Miller, 1989], is the error multiplied by a training factor β.

11.1.2 CMAC Discussion

The weight array is multi-dimensional, and so will be very large if k_s is larger than 3 or 4 and the components of the input vectors are quantized to more than 10 or so levels. This problem can be solved for CMAC by hashing the weight indices down to a much smaller array. Thus each cell in the smaller array will correspond to a very large number of cells in the original weight array. The hash collisions have been observed empirically not to cause great inaccuracy. This can be explained by distribution of increments over a neighbourhood: if a proportion of the distributed weights are corrupted, their expected summed error is small compared with the summed values of the valid weights.

CMAC has been used in recent work [Miller *et al.*, 1987] and has performed fairly accurately. This work needed only to learn a one-dimensional "strand" of the inverse dynamics of a robotic manipulator along a repetitive joint space trajectory. The concentration of data around this strand helped minimize hash-collision problems, providing good performance even with surprisingly small hashed-arrays of only 2000 elements.

A further advantage of CMAC is that it can learn when it is not provided with explicit

$$(\text{input}, \text{output}) \tag{11.3}$$

pairs, but simply rough

$$(\text{input}, \text{direction of prediction error}) \tag{11.4}$$

observations. This can be used to learn mappings such as evaluation functions in which an explicit evaluation of a state can rarely be obtained even after observation, but an indication of whether the current value is too optimistic or pessimistic *is* generally available.

CMAC's problems include the need to quantize, the need to keep neighbourhoods small (for efficient update and access), the inaccuracies caused by hash collisions and the difficulty of a formal proof of convergence.

11.1.3 Symbolic CMAC

An explicit geometric representation of the data passed to CMAC can have exactly the same prediction as CMAC itself. I call this new representation "Symbolic CMAC".

The representation is the explicit set of all the (input vector , increment) pairs we have experienced:

$$\mathbf{E} = \{(X_0, \delta_0), (X_1, \delta_1) \ldots (X_{N-1}, \delta_{N-1})\}. \tag{11.5}$$

To predict, the following calculation is used. $J_N(X)$ denotes the prediction at vector X after N observations.

$$J_N(X) = \sum_{(X_i, \delta_i) \in \mathbf{E}} \delta_i \frac{\mid nhood(X_i) \cap nhood(X) \mid}{\mid nhood(X_i) \mid} \tag{11.6}$$

This prediction can be computed naively by initializing the result as 0 and then for i between 0 and $N-1$:

1. Enumerate all the indices in $nhood(X_i)$.
2. Enumerate all the indices in $nhood(X)$.
3. Count the number of duplicate indices.
4. Add to the result: $\delta_i \frac{\text{number of duplicates}}{\text{number in } nhood(X_i)}$.

The proof that $J_N = I_N$ is by induction on N.

This rather inefficient implementation can be improved by working out the size of the intersections of neighbourhoods indirectly instead of in this explicit manner. In [Albus, 1975a] the neighbourhoods are designed to

1. Have the property that any two close input vectors will have a large intersection of neighbourhoods, and further vectors will share correspondingly less.

2. Contain a sufficiently small number of indices for acceptable performance.

CMAC's closeness measure is an approximation to Hamming distance. To facilitate computation the symbolic method will use instead the L_∞ metric. That is

$$dist(X,Y) = \max_i \mid X_i - Y_i \mid . \tag{11.7}$$

We can now define the neighbourhood of X as those indices within a fixed distance d of X (this is exactly what the CMAC method of choosing indices does in the one-dimensional case, and what it approximates in higher dimensions). The size of a neighbourhood is $(2d)^{k_s}$.

$$nhood(X) = \{Z : dist(X,Z) < d\} \tag{11.8}$$

The size of the intersection of two neighbourhoods of X and Y can now be computed. If $dist(X,Y) \geq 2d$ it is zero. Otherwise it is the volume of a k_s-dimensional cuboid. The length of the ith dimension of this cuboid is the distance of overlap in the ith dimension which can be shown to be $2d - \mid X_i - Y_i \mid$. The volume of the cuboid is thus

$$\prod_{j=0}^{k_s-1} (2d - \mid X_j - Y_j \mid) \tag{11.9}$$

Thus, for this metric, the prediction is

$$J_N(X) = \sum_{X_i \,:\, dist(X,X_i)<2d} \delta_i \; scale(X,X_i) \tag{11.10}$$

where

$$scale(X,Y) = \prod_{j=0}^{k_s-1} (1 - \frac{\mid X_j - Y_j \mid}{2d}). \tag{11.11}$$

Only those vectors lying within distance $2d$ from the input vector are inspected. If the vectors are stored in a balanced kd-tree, the expected cost of obtaining these vectors is $O(k_s(\log N + S))$ where N is the total number of vectors, S is the number of vectors which *do* lie within range and k_s is the dimensionality of **S** space.

The L_∞ metric was used simply because the neighbourhood intersection computation is easy. The L_2 (Euclidian) or L_1 (Hamming) metrics might have been preferred. The L_2 computation is the intersection of two hyperspheres, which is straightforward (and because the intersection size is simply a function of the distance between the two points it can be precomputed in a one-dimensional look up table). The L_1 calculation should also not be computationally expensive, but I have not attempted the unpleasant geometry to calculate the intersection.

11.1.4 Symbolic CMAC: Discussion

In this section I will mention some of the advantages of symbolic CMAC, but the main point of interest is not to suggest an alternative implementation, but the link between the apparently unrelated connectionist and geometric representations.

The **advantages** are:

- It is unnecessary to choose quantization levels for the components of the input vectors. This is an advantage (i) in terms of accuracy and (ii) because it is not necessary to decide in advance which ranges of the vectors need highest resolution.
- The neighbourhoods can be large without a dramatic increase in computational cost: for example they can consist of all points which have all components within 30% of the full range of input vector values. For the original CMAC this would mean an exponential increase in the number of indices in the neighbourhood for each new dimension. To avoid this, CMAC must use a fixed neighbourhood which only approximates the generalization that would be obtained from the Hamming distance metric.
- There are no inaccuracies induced by hashing collisions. This will be most important when the dimensionality is sufficiently high that each cell of CMAC's hash array would correspond to many weights *and* an adequate function approximation is desired over more than a small strand of the space. Without hash collisions the predicted function looks smooth, which may be of use if a gradient estimate is desired.

The **disadvantages** are:

- The memory size is not fixed. It is argued in Section 5.5 that this is acceptable because there will not be time for the memory to get large beyond current physical storage. However, the fixed memory requirement of CMAC, even if large, is more appealing.
- The time for access might be worse. The time to store a new observation in the symbolic CMAC is trivial, but accessing involves a range search. The cost of this search depends critically on the distribution of the data and the size of the neighbourhood. It is certainly bounded above by N, the number of observations, but in general can be expected to be very much cheaper. The author has not undertaken a detailed empirical comparison. The cost of Symbolic CMAC is expected to be similar to that of SAB learning, which in Section 10 is demonstrated as adequate.
- The original CMAC was partially motivated as an attempt to model the behaviour of the human cerebellum. It is clear that Symbolic CMAC is a large step away from the biological model, despite having the same behaviour.

11.1.5 Comparing CMAC with SAB Learning

The intended use of CMAC is the same as that of the nearest neighbour generalization: non-parametric prediction from data. It has been effectively demonstrated, particularly

by [Miller *et al.*, 1987]. It is shown here to be equivalent to a geometric representation, with an prediction obtained from basis functions. The basis function is $scale(X, Y)$ used in Equation 11.10, and now it becomes apparent that the choice of local weighting as being the volume of the intersection of two neighbourhoods is only one of a range of possibilities.

The disadvantages over the nearest neighbour are that its behaviour is very sensitive to the size and shape of the neighbourhood and that unlike the nearest neighbour there is no reasonable prediction far from any experience (CMAC will predict 0; nearest neighbour will provide at least a rough estimate). In a high dimensional space in which the data is necessarily sparse this is a serious problem except in familiar subregions. Nearest neighbour also provides an estimate of how reliable its prediction is, as the distance of the nearest neighbour.

Both CMAC and nearest neighbour have the problem of determining the scaling for the variables. The solutions proposed for nearest neighbour based on statistical analysis of the data could be applied equally well to CMAC.

CMAC and Symbolic CMAC are both resistant to a noisy and slowly changing environment. Naive nearest neighbour performs badly with such data, but the modifications used for *SAB* learning solve these problems. The most important advantage of CMAC and Symbolic CMAC is that they can still learn when trained not on input–output pairs but values indicating the direction of the error of the current prediction.

11.2 Reinforcement Learning using Dynamic Programming

Here we are concerned with a dynamic system which has a continuum of actions and a continuous two-dimensional state space. The task is of the class which needs a non-local control strategy, and so learning the world model is not the only problem.

The main thesis of this work has been that with abstractions provided by world models, other aspects of the controller may be preprogrammed with ease, and need not be learned. In this section, however, I discuss a simple investigation which considers one way of further increasing the controller's autonomy. The method is based on dynamic programming (DP) which is, for example, described in [Burghes and Graham, 1980]. Dynamic programming has recently been popularized as a method to be used in conjunction with reinforcement learning by [Sutton, 1990]. Figure 11.1 shows a simple dynamic system.

A puck is sliding on a bumpy surface. It can thrust left or right with a maximum thrust of one Newton in either direction. However, because of gravity, the actual horizontal acceleration varies. In fact, in the puck position shown, the maximum right thrust is not strong enough to accelerate up the slope. This is made more clear in Figure 11.2. It is a *state space diagram.* The puck's state has two components (its perceived position and velocity) and thus can be represented by a two-component vector. In the figure the horizontal axis corresponds to puck position and the vertical axis corresponds to puck speed. To explain further, the west of the diagram are states in which the puck is to

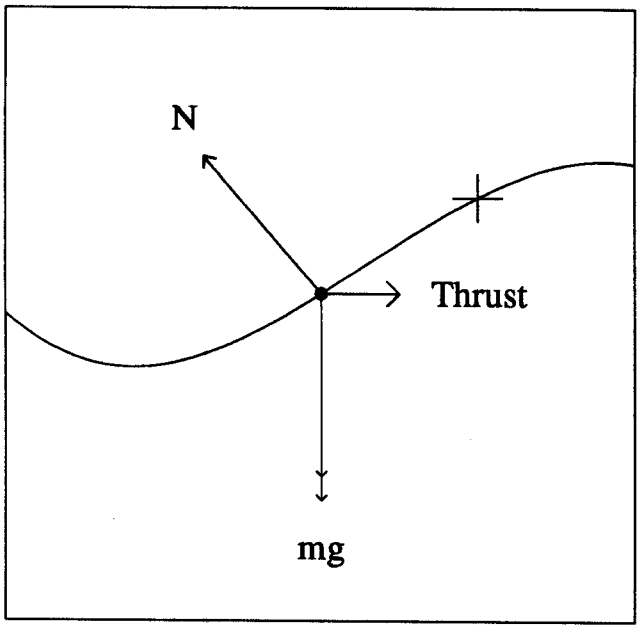

Figure 11.1

A frictionless puck acted on by gravity and a thruster. The target position is shown by the cross hairs.

the left, east denotes "puck on right", north denotes "puck moving rightwards" and south denotes "puck moving leftwards". The cross-hairs show the goal state, which is stationary, three quarters of the way up the hill (that is, state $x = 10\text{m}, v = 0\text{ms}^{-1}$, where x denotes position and v denotes velocity).

The arrows show the next state of the puck if it were to thrust rightwards with the maximum legal force of one Newton. Notice that at the centre of state space then even when giving this maximum thrust the steep slope causes the puck to gain negative velocity and slide left. Does this mean that if the puck is stationary at $x = 0$ that it is impossible to get to the goal state?

If the world model is known then the answer can be obtained by searching breadth first through all sequences of actions applied starting at the initial state. The more computationally tractable procedure of Dynamic Programming can be used to provide the same information. The dynamic programming algorithm is shown in Table 11.1.

To perform DP with a continuum of state and action variables it is necessary to partition both the state space and the action space into a finite number of possibilities. In this puck example the partitioning is achieved by quantizing the actions to 15 different levels and by quantizing each state variable to 64 levels, producing 4096 discrete states.

Dynamic programming is then performed, using, as the state transition function the analytic world simulation model. The results are portrayed graphically in Figure 11.3. They indicate that it *is* possible to reach the goal state by moving left initially and then accelerating to the right to gain sufficient speed to reach far enough up the hill before

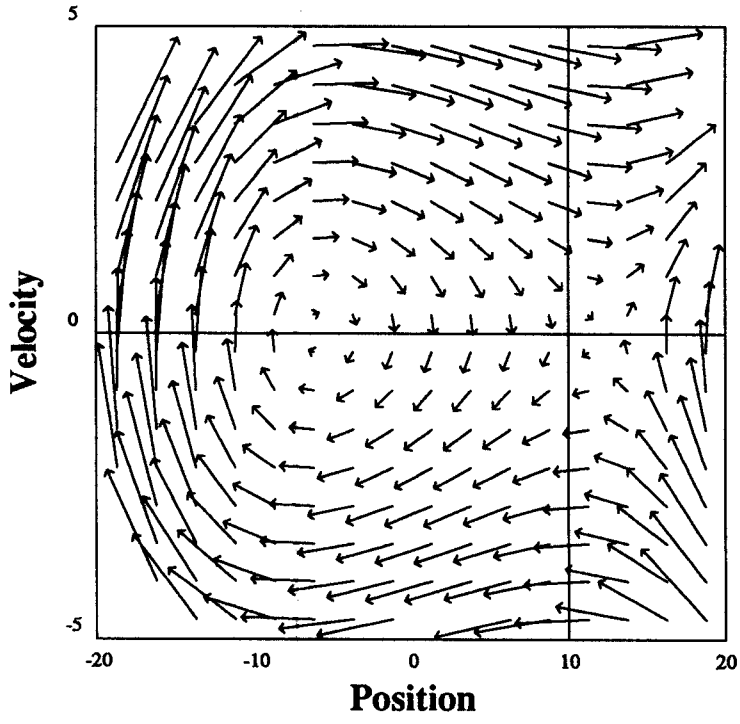

Figure 11.2

The state transition function for a puck which constantly thrusts right.

Algorithm:	Dynamic Programming
Input:	Finite set of states $\mathbf{S} = \{s_1, \ldots, s_n\}$ Finite set of actions $\mathbf{A} = \{a_1, \ldots, a_n\}$ State Transition Function $f : \mathbf{S} \times \mathbf{A} \rightarrow \mathbf{S}$ Set of goal states $\mathbf{U} \subseteq \mathbf{S}$
Output:	A set of triples $\mathbf{O} = \{(s_i, a_i, E(s_i))\}$ where $\mathbf{O} \subseteq \mathbf{S} \times \mathbf{A} \times$ **Integers**
Pre:	$\mathbf{U} \neq \phi$
Post:	The output triples have the following property. $(s_i, a_i, E(s_i)) \in \mathbf{O}$ if and only if $E(s_i)$ is the minimum number of time steps needed to get to any goal state starting from s_i and a_i is an optimal action which can be taken in state s_i to achieve this.
Code:	
1.	$\mathbf{n} := 0$
2.	$\mathbf{O_0} := \{(s_i, -, 0) : f(s_i, a_i) \in \mathbf{U}$
3.	while $\mathbf{n} = 0$ or $\mathbf{O}_n \neq \mathbf{O}_{n-1}$
3.1	$\mathbf{n} := \mathbf{n} + 1$
3.2	$\mathbf{O}_n := \mathbf{O}_{n-1} \cup$ $\{(s_i, a_i, \mathbf{n}) : (f(s_i, a_i), a_j, E_j) \in \mathbf{O}_{n-1}$ for some $a_j, E_j.\}$
4.	$\mathbf{O} := \mathbf{O_n}$

Table 11.1: The Dynamic Programming Algorithm

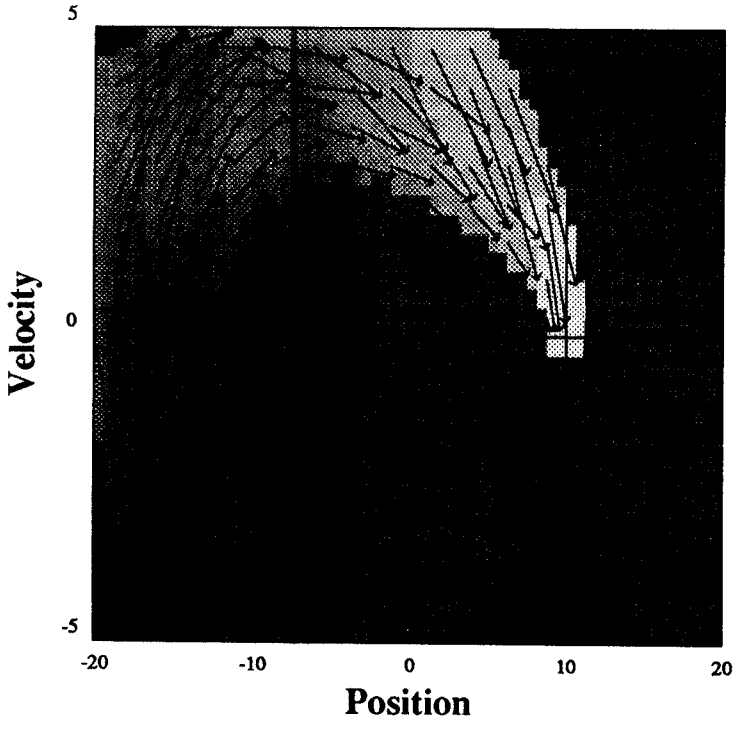

Figure 11.3

The brightness level in this diagram denotes the evaluation that DP gives the states. The arrows show the next states if you apply the actions recommended by DP. Notice that the evaluation function has a major discontinuity.

gravity brings the puck to a halt.

We are interested in what can be done if there is no predefined analytic world model to consult. Consider the case where the only experience of the world had been gained from the two trajectories shown in Figure 11.4, which consist of a total of 33 experiences (this was obtained by running two puck trials in which, on each control cycle, the puck chose a random thrust in the range -1 to 1 Newton).

A *SAB* world model model was learned from these experiences:

$$\underbrace{\text{Puck Position and Speed}}_{\textbf{State}} \times \underbrace{\text{Puck Thrust}}_{\textbf{Action}} \rightarrow \underbrace{\text{Puck Acceleration}}_{\textbf{Behaviour}} \tag{11.12}$$

Dynamic Programming was then performed using only this learned world model. The results are shown in Figure 11.5. The strategy obtained looks encouragingly similar to the optimal strategy obtained by DP with the analytic model, although there are errors, particularly in the right hand half of the state space (for example the bottom right hand corner). In fact there is also a region of critical error among the states leading up to the goal. Figure 11.6 shows the results of running using actions recommended by this DP's action map. Unfortunately the misplaced discontinuity leads to a missing of the target. This is not very disheartening because it was based on absolutely no experience near the goal state. Having had this failed attempt more information near the goal has been obtained. When this extra information is added to the *SAB*-tree, and DP is applied again with the updated world model, a superior controller is obtained (shown in Figure 11.7,

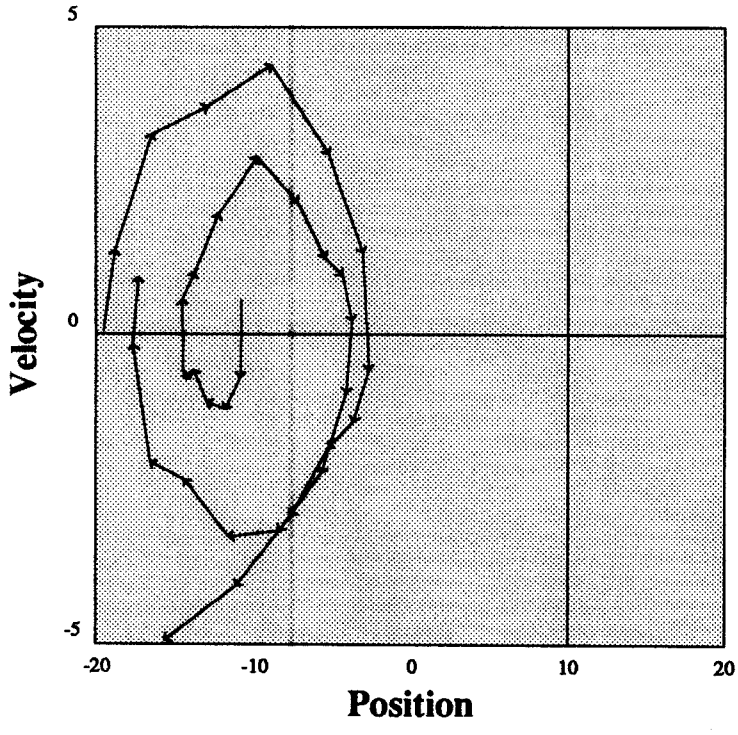

Figure 11.4

Some random experiences in the Puck's state space.

though the difference is unfortunately hard to detect visually) which is now sufficient to achieve the goal on the next attempt (see Figure 11.8).

It has thus been shown that the world models obtained from *SAB*-learning can be used instead of an analytic model to autonomously compute non-local control. However, DP works by quantizing and enumerating the state and action spaces which is exactly what the *SAB* world model learner has been designed to avoid. Thus it seems a shame that the world model learner can survive high dimensions but the controller which uses it cannot. Even with the fairly trivial Puck example, DP is rather computationally expensive (five minutes processing on a UNIX workstation using the *SAB*-learned world model).

11.2.1 Variable Resolution Dynamic Programming

Instead of enumerating the state and action spaces, let us follow the same approach that was used for *SAB*-learning. Here, an experimental method is described which performs computation only within those areas of state and action space which have actually been experienced.

First, let us consider in depth the assumptions that DP makes in order to estimate the minimum time to goal. We shall denote this estimation by E_{ttg}.

1. $E_{\text{ttg}}(s_1) \leq E_{\text{ttg}}(s_2) + 1$ if the world model predicts that for some action it is possible to get from state s_1 to state s_2 in one time step.

2. $E_{\text{ttg}}(\text{Goal State}) = 0$

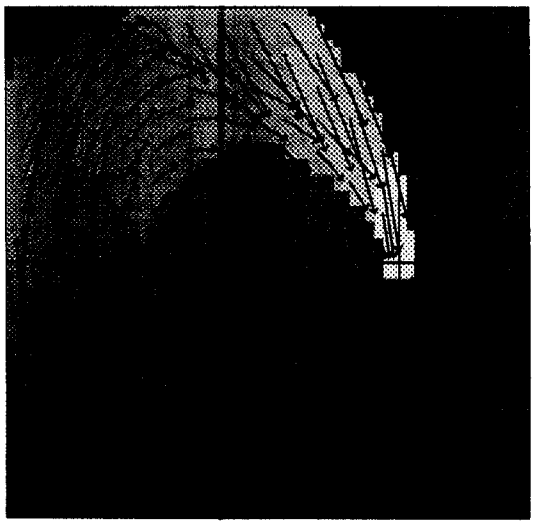

Figure 11.5: Dynamic programming using a world model obtained from *SAB* learning.

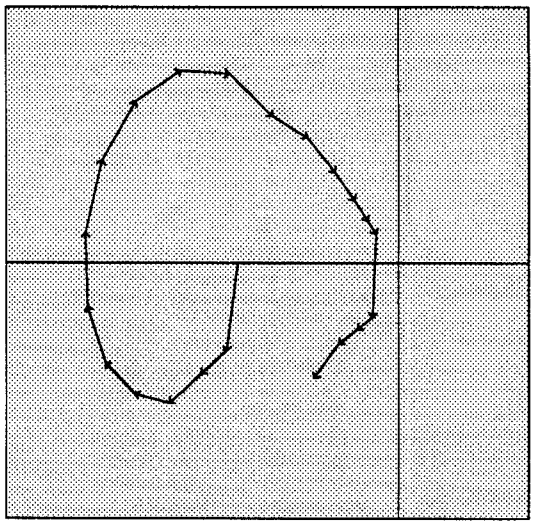

Figure 11.6: Trajectory produced by using the results of Figure 11.5.

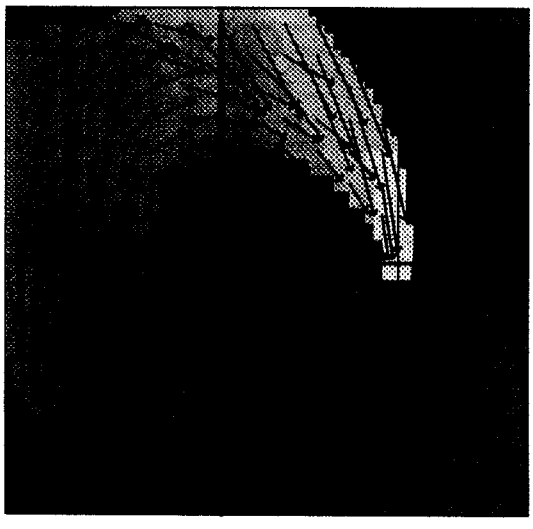

Figure 11.7: Dynamic programming using a learned world model updated with the experience of Figure 11.6.

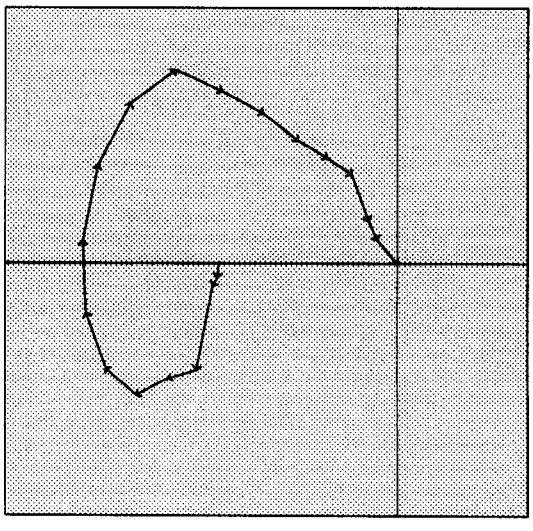

Figure 11.8: Trajectory produced by using the results of Figure 11.7.

3. The estimated time to goal (E_{ttg}) for all states in the same quantization cell is the same.

The first and second assumptions are logically valid, and the third assumption is necessary in order to supply DP with a finite number of states. This is not necessarily accurate, but accuracy can be gained by increasing the number of quantization levels, and hence the number of distinguishable states. It is based on the assumption that E_{ttg} is smooth and will thus not vary significantly within each quantization cell.

The smoothness assumption is not necessarily true (for example, there is a discontinuity in Figure 11.3), but away from discontinuities provides an excellent estimate of the optimal action.

To perform the equivalent of DP without needing to look at states other than those that we have experienced, let us make an alternative smoothness assumption:

3a. E_{ttg} has maximum slope A. For any two states s_1 and s_2

$$\mid E_{\text{ttg}}(s_1) - E_{\text{ttg}}(s_2) \mid \leq A \mid s_1 - s_2 \mid \tag{11.13}$$

The new DP algorithm evaluates states by finding the most pessimistic E_{ttg} which agrees with the world observations and which fits the constraints 1, 2, and 3a above.

Why is the most pessimistic value within the constraints used? The advantage is that it is the most clearly defined interpretation. A moment's reflection shows that the optimistic evaluation within constraints would be useless: it would simply be that $E_{\text{ttg}}(s) = 0$ for all states. There is also a computational advantage in being pessimistic—the E_{ttg} estimate can then be computed for each state that has been experienced with reference only to other states which have been experienced.

The only states which have their E_{ttg}'s recorded are the goal state(s) and all states which have been experienced. The latter are recorded in a *SAB*-tree.

Initially, before any experience, the estimated time of the goal state (s_{goal}) is zero, and by (3a) we know that for any other state s,

$$\mid E_{\text{ttg}}(s) - E_{\text{ttg}}(s_{\text{goal}}) \mid \leq A \mid s - s_{\text{goal}} \mid \tag{11.14}$$

Thus, because we use the most pessimistic estimate, and putting $E_{\text{ttg}}(s_{\text{goal}}) = 0$, we derive

$$E_{\text{ttg}}(s) = A \mid s - s_{\text{goal}} \mid \tag{11.15}$$

This produces Figure 11.9. The apparent quantization is due to the primitive 2-d graph drawing software. There is no quantization for the actual algorithm. Here, A is chosen as 6.4 time steps per metre, which corresponds to gradient 500 in uniformly scaled *SAB*-tree units. The start state ($x = 0\text{m}, v = 0\text{ms}^{-1}$) has $E_{\text{ttg}} = 64$ time steps.

I shall now describe the operation of finding the most pessimistic E_{ttg} that satisfies constraints (1), (2) and (3a). We are given a known goal state s_{goal} and a set of observations

$$\{s_0 \rightarrow s_0', \ldots, s_n \rightarrow s_n'\} \tag{11.16}$$

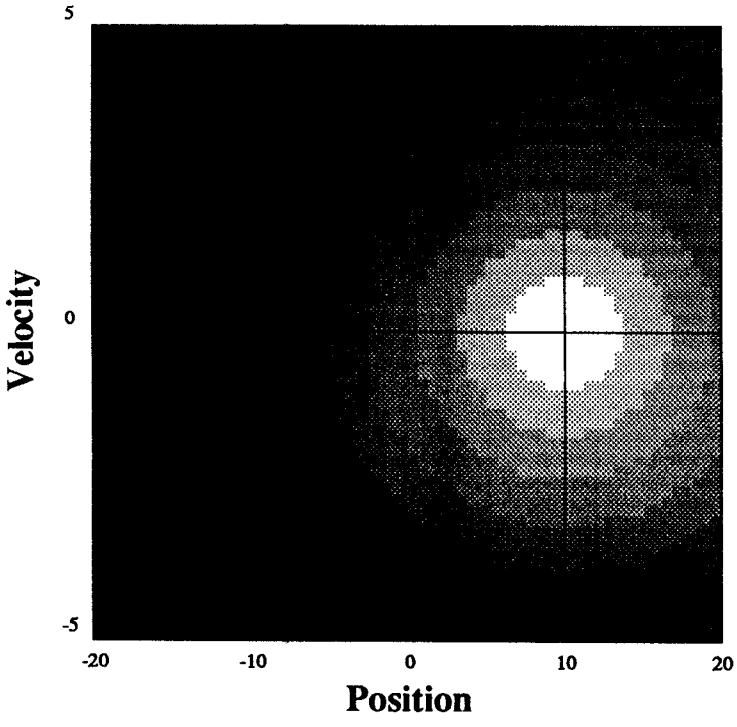

Figure 11.9

E_{ttg}—the pessimistic, slope bounded evaluation function after no experience. The only knowledge is that the target state is zero time steps to success.

The operation is achieved by two algorithms. The first, **Evaluate**, finds the value of any state, given the known values of a set of other states, by means of constraint (3a). The second algorithm computes the E_{ttg} for all experienced states $s_0, s_0', \ldots, s_n, s_n'$. It uses the **Evaluate** algorithm and constraints (1) and (2). The reader who is interested in the behaviour of these algorithms, but not their implementation, should skip to Section 11.2.4.

11.2.2 The "Evaluate" Algorithm

This takes, as input, the query state s and a set of known evaluations

$$\mathrm{ES} = \{(s_0, E_0), \ldots, (s_k, E_k)\} \tag{11.17}$$

Let us call such a set an *evaluation set.*

The simplest algorithm to estimate $E_{ttg}(s)$ would look at each member of ES in turn. Constraint (3a) tells us that for each member $(s_i, E_i) \in \mathrm{ES}$, $E_{ttg}(s) \leq E_i + A \mid s - s_i \mid$. In particular we know that $E_{ttg}(s)$ is no greater than this expression for the value of i which minimizes $E_i + A \mid s - s_i \mid$. To be pessimistic, it must be assumed that $E_{ttg}(s)$ equals this minimum value.

The algorithm of finding the minimum value by scanning the whole evaluation set would be valid, but slow. A superior algorithm has been developed which searches the evaluation set as a *k*d-tree in a similar manner to that of a nearest neighbour search. Search states are often reached at which the lowest value of $E_i + A \mid s - s_i \mid$ yet found is sufficiently low that other hyperrectangles in the tree need not be searched. This occurs

when the distance d of the hyperrectangle to the query point is sufficiently great, that the lowest possible expression of any point in it (which is $0 + Ad$) is greater than the best yet found, and so search cutoff may occur. With a large value of A almost all the tree is cut away. The value of A in chosen as large anyway, in order to increase the chance the constraint (3a) is valid.

11.2.3 Evaluating all Experienced Points

Here I explain how to fill the set of all observed states with their correct E_{ttg} values, given a set of observed state transitions

$$\{s_0 \rightarrow s_0', \ldots, s_n \rightarrow s_n'\} \tag{11.18}$$

It starts with an estimate of the E_{ttg}'s of all states based solely on the distance to the goal state. It then iteratively reduces this estimate using (1) and (3a) while ensuring the estimate never becomes more optimistic than is necessary to satisfy the constraints.

An evaluation set called ES is constructed.

$$\text{ES} = \{(s_0, E_0), (s_0', E_0'), \ldots, (s_n, E_n), (s_n', E_n')\} \tag{11.19}$$

It is initialized by

$$E_i = A \mid s_{\text{goal}} - s_i \mid \qquad E_i' = A \mid s - s_i' \mid \qquad \text{for every } i. \tag{11.20}$$

Then the following two steps are performed

1. For each i, $E_i := \min(E_i, 1 + E_i')$. This is necessary to ensure constraint (1) is upheld. A state cannot have an evaluation more than one greater than a known successor state.

2. All the E_is and E_i's are all recalculated using the **Evaluate** algorithm applied to the evaluation set ES. This is necessary because the previous operation might have made some previously poor states into good states. Other states which are close to these improved states should themselves be given superior evaluations to maintain constraint (3a).

The two operations above are repeated in turn until there are no further changes to ES. Although not proved here, it is believed that, as is the case with the standard Dynamic Programming algorithm, this termination only occurs when all constraints are satisfied, and that the resulting evaluation is the most pessimistic.

11.2.4 Algorithm Behaviour

To illustrate this procedure, let us consider the results of incorporating a small number of transitions. The trajectory and the resulting pessimistic evaluation function are shown

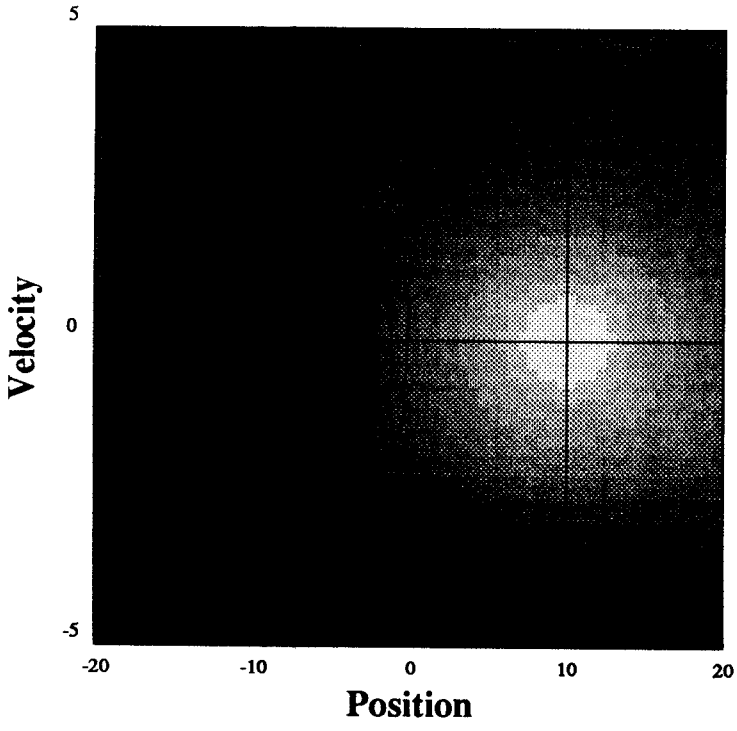

Figure 11.10

Pessimistic slope bounded evaluation after a little experience.

in Figure 11.10. Notice that the evaluation has improved for the states early on which subsequently followed an improving trajectory but did not change for the part of the trajectory which became worse.

After further experiences, some random and some using the evaluation function (unsuccessfully), the experiences and evaluation function looked as in Figure 11.11. It can be seen to resemble the evaluation function obtained by normal dynamic programming. This is further confirmed by inspection of Figure 11.12 which shows the action map derived from the evaluation function. However, it can also be seen that unless a particularly lucky random experiment takes place, no trial starting at the state space centre will succeed. Following the arrows in Figure 11.12 it can be seen that it will first travel down and to the left through state space, then up, on the left side until it swings rightwards moving along the top. However it comes down too early, meaning it slows down too soon and doesn't make it up the hill. Thus it gets close, but not close enough. This can be seen to have happened already several times in Figure 11.11. This performance is good, considerably better than with the naive local controller, but not quite good enough.

Rather than wait for a lucky random experiment to occur, I gave the controller a hint by starting the puck in a very good state, which was still scored badly by this evaluation function. This is the state of moving rightwards at great speed at $x = 0$. It is good because the hill can be used to decelerate in time to reach the goal. The interesting question was whether the controller would generalize from this extra success.

The trial was carried out, causing the evaluation function shown in Figure 11.13. This

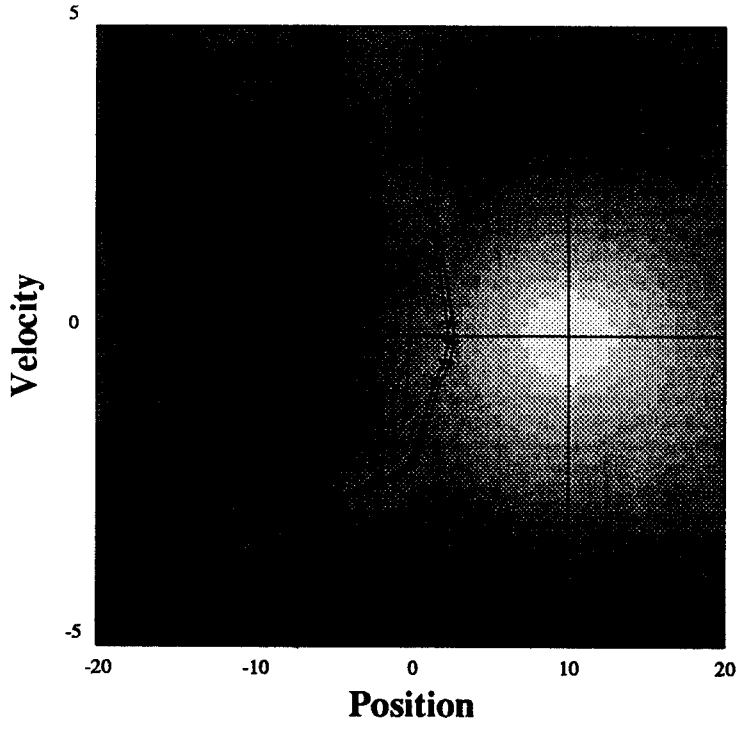

Figure 11.11

$E_{\mathbf{ttg}}$, the pessimistic slope bounded evaluation after further experience.

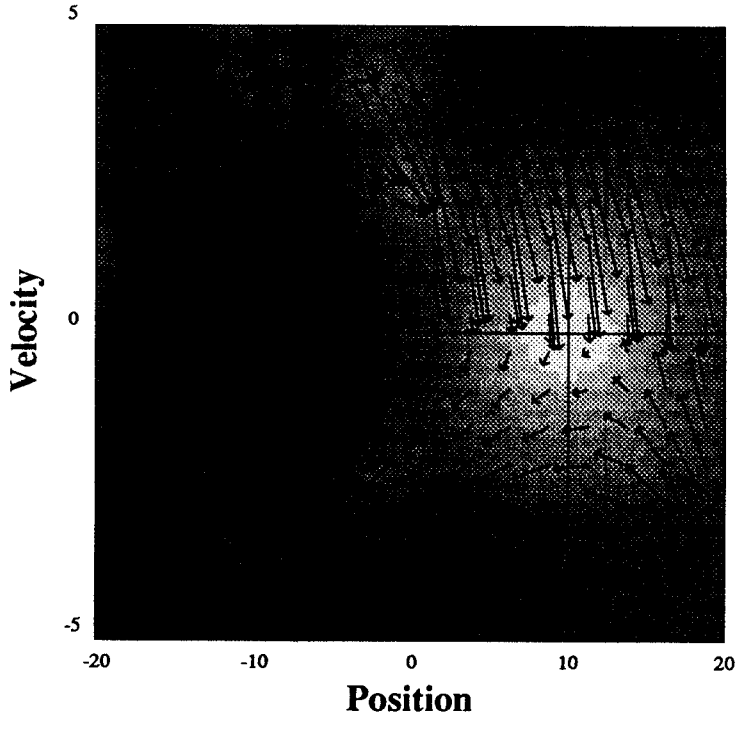

Figure 11.12

The thrusts recommended by the evaluation function of Figure 11.11.

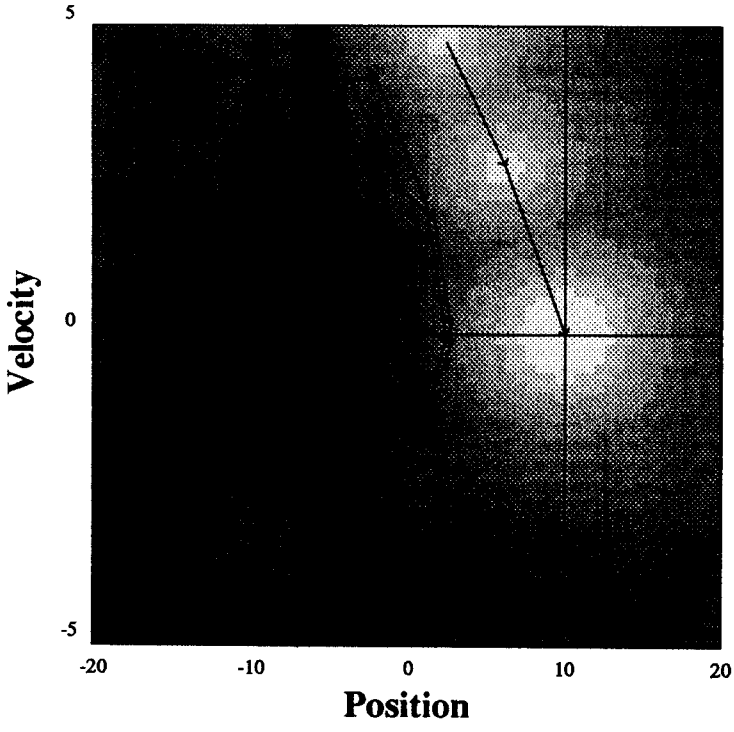

Figure 11.13

E_{ttg}, the pessimistic slope bounded evaluation after further experience, including starting at the state $x = 5, v = 4.5$. This start state lead easily to the goal.

new evaluation function is good enough to aid the control of the puck from the centre of state space, as shown in Figure 11.14. This is encouraging because some useful experience in one area of the state space has aided entirely different areas—a very non-local effect.

11.2.5 Discussion

The initial Variable Resolution Dynamic Programming implementation has demonstrated that it is possible to learn to control dynamic systems using non-local strategies. Its advantages are:

- **Variable resolution.** It can adjust to resolutions of interest and is non-parametric. These two important features have been discussed earlier in this dissertation (see Section 5.2).

- **Inexpensive computation.** The cost of updating the evaluation function is low. It happens once at the end of each trial, though there would be nothing to prevent it from occurring incrementally, in parallel with the trials. The typical update cost is $N \log N$ in the number of experiences because the number of dynamic programming iterations is usually only one unless a great "discovery" is made which leads to re-evaluation of non-local parts of the state space. This is a considerable improvement on high dimensional Dynamic Programming.

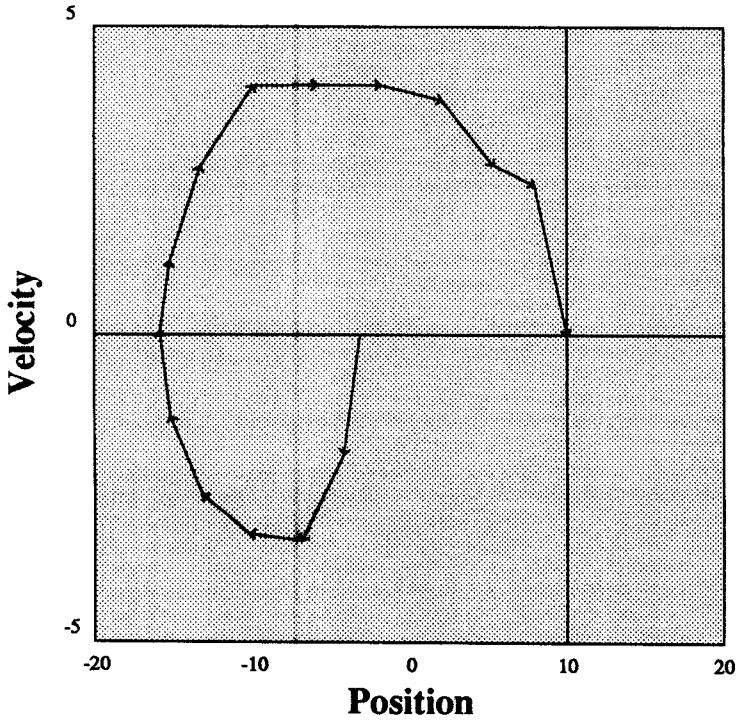

Figure 11.14

The trajectory followed by means of the evaluation function of Figure 11.13.

- **Learning rate.** The rate of learning is high compared with non-DP methods because the one-shot property of *SAB* learning is retained.

However, the initial implementation still has some serious problems.

- **Serious discontinuities** seem likely in many evaluation functions. The smoothness assumption (**3a**) is invalid. Empirical evidence is needed to determine how seriously this undermines performance.

- **Gaining useful experience.** The problem of gaining experience is more difficult than it was for the *SAB* action chooser because those areas of the state space in which it is important to experiment may themselves be very hard to reach. Furthermore, in a dynamic system such as this, random actions usually result in very quick disaster.

- **Not proven.** The only domain in which it has been attempted is a two-dimensional state space in which quantization of variables is actually adequate. A more complex test is required.

It is not clear that it is important to learn non-local or optimal controllers. The example task here could have been controlled by a slightly more complex local controller which used the tactics

1. Get to a stationary state at the bottom of the valley.

2. Move to $x = -18$m.

3. Accelerate towards the goal.

Such a controller is slightly more work on the part of the system designer, but is not difficult to achieve. It seems likely that to solve the non-local control autonomously it might be more profitable to automate the acquisition of qualitative plans such as that above, rather than to compute them at a low level as we have been doing in this section.

Chapter 12

Conclusion

The conclusion takes the form of a summary of the work of the dissertation, then a description of its important contributions, and finally a discussion of possible extensions.

12.1 Summary

The principal theme of this dissertation has been that realistic robot learning can occur if the things which we choose to learn are models of the world, and that this is particularly useful if the learning is practical, efficient, fast and robust. The supplementary theme has been that learned world models are sufficient to make the rest of robot control considerably easier.

A robot task is controlled by the *SAB* control cycle, reproduced for a final time in the left of Figure 12.1. World models are learned by explicitly storing every experience as a triplet of data: the perceived state in which the experience occurred, the raw action which was then applied, and the resulting perceived behaviour.

By generalizing to new experiences using the closest known experience, it has been explained and demonstrated that learning happens quickly.

Part of the dissertation has dealt with the data structures and algorithms necessary to ensure robust nearest neighbour performance. These have included mechanisms for fast access and update, and robustness to disorder. The dissertation also provides a theoretical analysis of what can be learned, and how quickly, by the nearest neighbour method. The resulting data structure is the *SAB*-tree: a modified, data-sharing, pair of *k*d-trees.

The *SAB* action chooser performs more slowly during early learning, when it knows that it does not know how to achieve requested behaviours. In such cases it can choose the most promising and informative out of a variety of candidate actions, by means of the P_{succ} probability of success heuristic, which is defined in terms of the partially learned forward (**State** $\times$ **Action** $\rightarrow$ **Behaviour**) world model. The first candidate action considered by the *SAB* action chooser is always the one recommended by partial inversion. This is

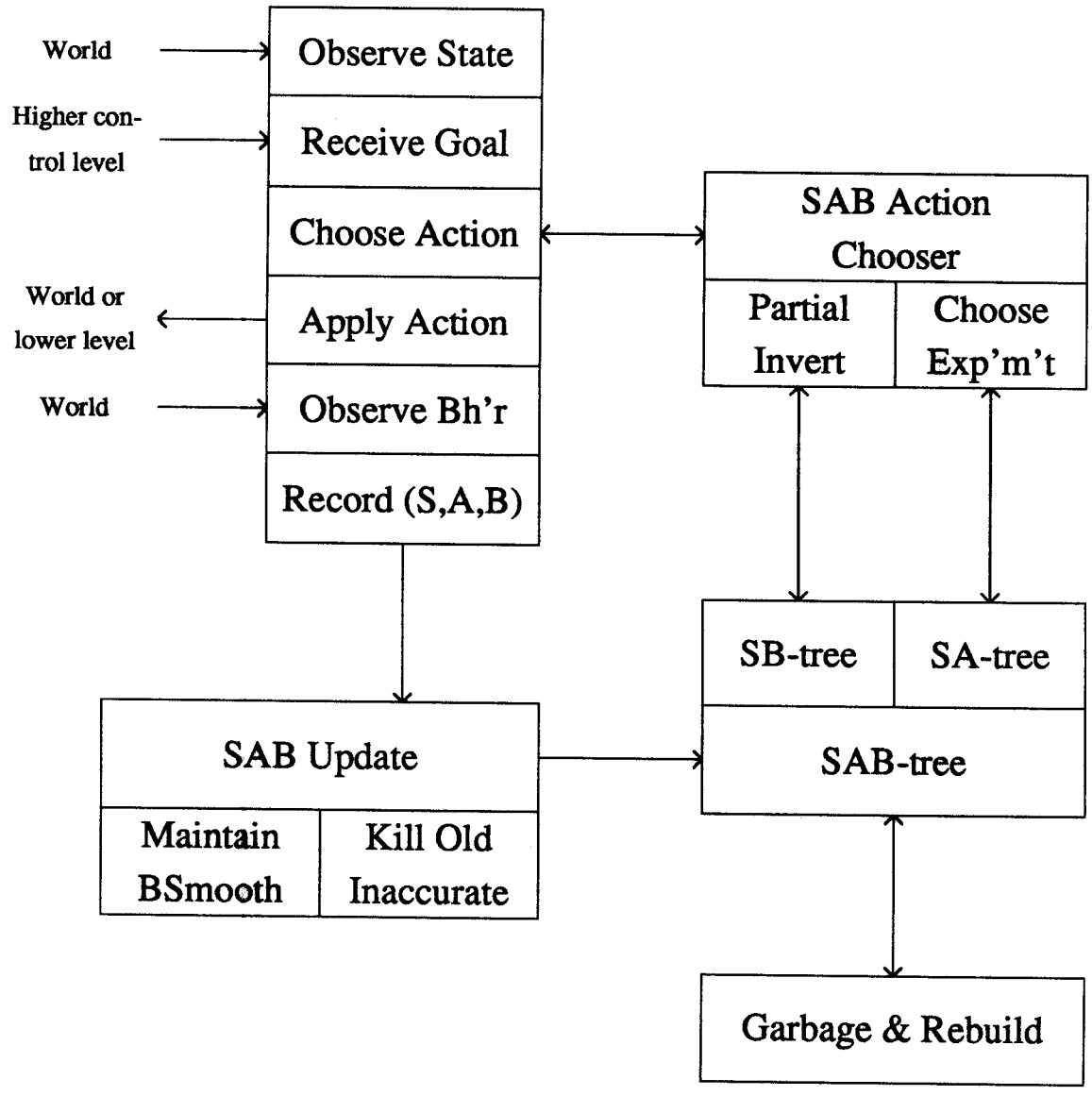

Figure 12.1

The structure of *SAB*-learning.

obtained by nearest neighbour search of the inverse (**State** $\times$ **Behaviour** $\rightarrow$ **Action**), and possibly faulty, model. This is checked by passing it through the forward model, and if adequate the partial inversion action is used. As more is learned, access becomes faster because the partial inversion estimate is usually known to be suitable, and thus almost no action choosing computation is required. The system becomes increasingly reactive as task performance improves.

An important feature of this work is the attempt to escape the curse of dimensionality. This can only be done in one of two ways.

1. Assume there exists extra structure in the problem and then find the best fit of the data to the structure. This is effective if it is known that any new problem *will* have a solution in the required structure. Otherwise learning can get stuck.

2. Deliberately restrict learning to particular low dimensional subsets of the data which are sufficient to solve the task. This approach was adopted.

The dissertation then describes how

- The part of robot control which is conventionally regarded as the bottom level—the modelling of the robot/world dynamics—can be achieved automatically by *SAB* learning. Furthermore the models can be learned for *any* perceived variables sufficient to determine the state, action and behaviour of the robot.

- Middle abstraction levels such as stability and trajectory tracking can be dealt with in a unified manner—ice puck control. These are tasks in which the behaviour required can be obtained very easily from the current perceived state and current perceived goal.

- Complex tasks, for which local perception-based control strategies do not exist, can be broken up into a hierarchy of simpler *SAB* controllers at varying levels of abstraction. This is demonstrated for the difficult control task of volleying a ball into a bucket with a dynamic robot arm. At this stage the structuring needs to be performed by some expert, presumably human. In order to be able to classify the controller as truly autonomous, it would have to be able to devise the structuring strategy itself. At this stage the lack of full autonomy is a compromise, justified by the observation that a large proportion of the effort in robotic control using conventional methods is not in inventing abstract strategies, but in modelling the world.

12.2 Contributions

12.2.1 Design and Detailed Investigation of the SAB Control Cycle

This work has integrated perception, kinematic and dynamic learning, and posed them as one problem: the relationship between raw perceptions of state, action and behaviour. It was shown that much of the previous model-learning work in the field can be cast in the same format. The *SAB* control cycle was introduced, and its use was demonstrated by a range of detailed empirical experiments.

12.2.2 Using Nearest Neighbour to Learn World Models

The nearest neighbour method had been used for classification for some years. Its use to learn mappings had been suggested, but generally passed over in favour of asymptotically more accurate, though slower and less general, local regression methods. [Clocksin and Moore, 1989] is the first published work known to the author which actually uses the nearest neighbour method to learn a robot world model. Other independent investigations [Mel, 1989; Atkeson and Reinkensmeyer, 1989] have successfully used nearest neighbour within robot learning investigations, though in each case with a different end in sight.

This work has enhanced the *k*d-tree based nearest neighbour method for the purposes of *SAB* learning. In particular it has

- Modified it to simultaneously learn the valid "safe" forward model **State×Action → Behaviour**, and the useful (but not necessarily valid) inverse model **State×Behaviour → Action**. The forward model allows the correct model of the world to be learned without the danger of "sticking" or ambiguity dangers which the inverse model can cause. However, when the inverse model is valid it can be consulted for fast performance.

- The simple form of nearest neighbour is in danger of performing poorly in a noisy or non-stationary environment. This work therefore augments the method to cope with these situations in a robust fashion which is also optimized for efficiency. The extra cost is borne entirely at world model update time, which is argued to be less critical than at world model access time because (i) accesses are more common than updates, and (ii) if absolutely necessary the world model could be updated off-line (as the STORE.AFTER experiment demonstrated in Section 10.3).

- The *k*d-tree based nearest neighbour algorithm was tested under a variety of conditions.

12.2.3 SAB Action Chooser

I have implemented a probabilistic mechanism which solves the perform/experiment dilemma by using knowledge about its own knowledge to decide whether experimentation is nec-

essary, and if so what a good experiment would be. I have shown how the process of repeated experimentation can be viewed as an optimization process which will not get stuck in local minima. In the neighbourhood of the optimal action, the error decreases exponentially with subsequent choices (the number of bits of accuracy increases linearly). The *SAB* action chooser also biases the dense, high accuracy, learning to take place only in restricted subregions of the control space, which is essential for learning to take place in high dimensionality.

12.2.4 Integration of *SAB* Control with Compound Controllers

I developed a technique for hierarchical learning to take place at multiple levels of abstraction which, unlike earlier proposals, does not suffer from the danger of being stuck due to the credit or blame assignment problem. This danger is avoided because the system learns, at all levels, only objective observations about the world, rather than action maps.

12.2.5 Incidental Contributions

- Based on empirical tests, I have improved the *k*d-tree building algorithm in order to increase nearest neighbour search speed, at the expense of some tree balance.
- I have provided a survey of recent work in learning robot control.
- I have extended the proofs of PAC-learnability produced by [Aha *et al.*, 1990; Kibler *et al.*, 1988] for nearest neighbour based classification, to the learnability of generally continuous functions.
- I have identified and proved the potentially useful relationship between Albus's CMAC [Albus, 1975a; Albus, 1975b] and a *k*d-tree-based representation—"Symbolic CMAC".
- I have made some initial demonstrations of how world model learning can also help autonomously learn non-local control strategies. This has been by means of dynamic programming, supplemented by a learned *SAB* world model, and by means of an experimental variable resolution version of dynamic programming.

12.3 Future Work and Extensions

The following areas were identified as important issues in Chapter 3, which surveyed the learning control field. Can this work be extended to encompass them?

12.3.1 State Identification

This work has assumed that it was an easy job for the system designer to specify which sensor inputs were sufficient to determine the system state. If this were not the case,

then what could be done? One possibility would be to let the robot use the values of all conceivable sensors and previous action signals—and perhaps their time derivatives as well. It is likely that all the information would be sufficient to determine state, but there would be a great deal of redundancy which would slow down all algorithms which were time dependent on state vector length, and also impair the nearest neighbour algorithm. Solutions to this have been proposed:

- In [Vogel, 1989], potential state signals which have no effect on behaviour are detected.
- In [Aha *et al.*, 1990], state components are weighted, and weights of "poorly predicting" components are gradually reduced to zero.
- In [Simons *et al.*, 1982], the most recent sensor readings are initially considered sufficient state representations, but in the presence of apparent non-determinism increasingly older sensor readings and action choices are used to augment the state.

12.3.2 Non-determinism

A new mapping must be learned in place of **State** × **Action** → **Behaviour**. This is

$$\textbf{State} \times \textbf{Action} \times \textbf{Behaviour} \rightarrow [0,1] \tag{12.1}$$

This is the relationship between the state, the action applied and the probability distribution of the behaviour.

The probability density can be estimated using an exemplar set **E** in a variety of ways [Omohundro, 1987]. The *SAB* action chooser is already designed to choose the action with the highest probability of success, but in its current implementation the uncertainty is only in terms of lack of information. The estimate could be supplemented with an additional term indicating the estimated inherent uncertainty in applying an action.

12.3.3 Induction

The *SAB* learning method could not be called inductive. The representation of the world model is not an attempted explanation of the data, instead it *is* the data.

This non-inductive representation is used precisely to avoid making extra assumptions about the data, which could lead to stuck learning were they incorrect.

Nonetheless, once the data were gathered it would be interesting, and possibly useful for more abstract aspects of control, to try to explain it. There are numerous inductive learning systems, including decision trees and genetic classifier systems, but a particularly suitable method might be Aha's instance-based learning [Aha *et al.*, 1990; Kibler *et al.*, 1988], which obtains a simple explanation in the form of a small number of exemplars under the nearest neighbour generalization.

12.3.4 Further Work on Autonomous Non-local Control

It is intended that variable resolution dynamic programming, as described in Section 11.2, be investigated further. If successful it might help take the automation of system design even further up the hierarchy of abstractions. For example, the acquisition of techniques such as ice puck control, might be automated.

12.3.5 Fast Local Regression

Some of the early work of this investigation used regression based on a selection of local exemplars instead of nearest neighbour prediction. This was abandoned because

- Nearest neighbour gave good performance accuracy for a considerably simpler algorithm.
- Local regression was very hard to implement with adequate real time performance (for reasons described in Section 5.2).
- Local regression was much less noise-tolerant.

However, with increasing computational power it might be easier to use local regression instead of, or in conjunction with, nearest neighbour, particularly to help guide the *SAB* action choosing search.

Further interesting and important extensions include additional empirical tests with real, dynamic robots, and also the intriguing question of how to learn "safely"—in environments where certain sorts of poor performance, even once, would be unacceptably expensive.

12.4 Concluding Remarks

Robotics is a young and promising discipline but it is generally agreed that robots would benefit considerably from more autonomy than that of present industrial machines. It is also well known that models of the behaviour of the robot can considerably simplify robot control. Here we have seen that a robot can learn surprisingly quickly and easily from its environment by simply recording everything that it observes. It is this central idea which has motivated the development of the *SAB* learning system, and it is to this idea that the success of these initial demonstrations must be attributed.

It is hoped that some of the ideas of this dissertation might prove useful in practical systems. It is an exciting thought that the acquisition of world models might be a realistic next step in robot automation.

Appendix A

Format of Graphs

Figure A.1 shows an example of the graph notation used in this dissertation. The graph was obtained from running, in this case, forty experiments (hence the "$N = 40$" on the key). For each x-coordinate on the graph, there were thus forty statistics. The y coordinate corresponding to each x-coordinate is the mean of these forty statistics. In order to represent the spread of the data, the standard deviation of each set of statistics is shown. The length of each bar above the mean point is the standard deviation. Occasionally some standard deviation bars are so large that they do not fit entirely on the graph and are thus clipped.

The standard deviations have a second use. Along with knowledge of the number of statistics, they can indicate the confidence that the mean of the sample is the true mean of the population. For N observations, the probability that the mean lies within $1.96/\sqrt{N}$ sample standard deviations of the mean is approximately 95% [Hoel, 1971]. This is because the sample mean μ_s of a reasonably large set of observations behaves like a normally distributed variable with mean μ_s and variance σ_s^2/N where σ_s^2 is the sample variance. The number of observations for this to be generally accurate has been observed as approximately thirty.

For example, if the mean value is 10, the sample standard deviation 5 and the sample size 40, there is a 95% probability that the true mean of the population is between

$$10 \pm \frac{5}{\sqrt{40}} \tag{A.1}$$

The proportional length of the confidence interval is shown to the left of the key.

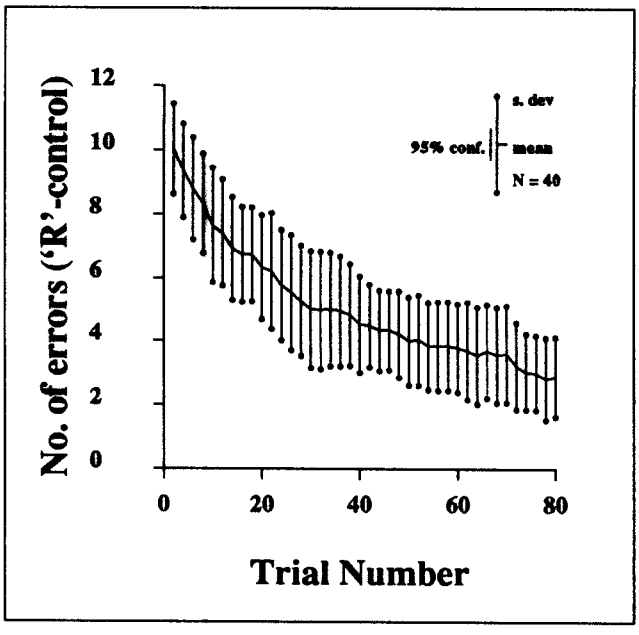

Figure 1.1

A graph typical of this dissertation.

Appendix B

Nearest Neighbour Polynomially Learns Continuous Functions

Looking back at the proof of nearest neighbour learnability given in Section 5.3, the number of exemplars required for the coverage lemma depends directly on ϵ, γ and α where in turn γ depends on β and ϵ depends on β and θ. We know that if we find an n such that (i) $n > N(\epsilon, \gamma, \alpha)$ as defined in Equation 5.12, (ii) $\gamma < \frac{1}{2}\beta$ and (iii) that $\mid I_{\epsilon_1} \mid > 1 - \frac{1}{2}\beta$, then (with probability $1 - \alpha$) we will have achieved tolerance θ.

Here we begin by showing, for the case of a uniformly continuous function, that the dependence of ϵ on θ is linear. This is because a uniformly continuous function has the property (see [Burkhill, 1978]) that

$$\begin{aligned} &\exists G \in \Re : \\ &\quad \forall x, x' \in [0,1]^k \quad \mid f(x) - f(x') \mid < G \mid x - x' \mid . \end{aligned} \tag{B.1}$$

Thus if we choose $\epsilon = \theta/G$ we know that for any domain points x,x'

$$\begin{aligned} &\mid x - x' \mid < \epsilon \\ \Rightarrow \quad &\mid f(x) - f(x') \mid < G \mid x - x' \mid < G\epsilon = \theta \end{aligned} \tag{B.2}$$

as required. Using this ϵ, the number of exemplars $N(\epsilon, \gamma, \alpha) = N(\theta/G, \beta/2, \alpha)$ which is

$$n = \left(\frac{2}{\beta}\left\lceil\frac{G\sqrt{k}}{\theta}\right\rceil^k\right) \times \log\left(\frac{1}{\alpha}\left\lceil\frac{G\sqrt{k}}{\theta}\right\rceil^k\right). \tag{B.3}$$

Thus when learning uniformly continuous functions the number of k-dimensional exemplars required is better than $(k+1)$-nomial in $\frac{1}{\theta}$, linear in $\frac{1}{\beta}$ and sublinear in $\frac{1}{\alpha}$.

It can also be shown that for bounded-slope functions with discontinuities caused by finite region boundaries, the dependence of ϵ on β and θ is polynomial (proved for a strongly related case in [Aha *et al.*, 1990]). Furthermore, it is conjectured that for asymptotes for which the gradient increases polynomially with the reciprocal of the distance to the discontinuity, that the dependence is again polynomial.

Appendix C

Estimating whether a Directed Behaviour is Attainable

The heuristic described in this appendix was used in the "autonomous time-choice" experiments in Section 10.4. The occasional need for such a heuristic was discussed in Section 8.3. An estimate is required of the probability that a behaviour is attainable at the current state. The task controller can use this probability to decide if it is sufficiently unlikely that it it not worth requesting of the *SAB* action chooser.

This probability is a heuristic estimate. With the same motivation as has the P_{succ} estimate (Sections 8.1 and 8.2), it is chosen to be robust and computationally cheap. It assumes the goal behaviour has a *direction*. This is equivalent to defining a behaviour origin $\mathbf{o}_B$. The concept of a behaviour having a direction is generally acceptable—for example the behaviour of a dynamic manipulator is an acceleration vector which has a clearly defined origin and direction. Thus, let us define the magnitude and direction of a goal vector by

$$M_{\text{goal}} = |\, \mathbf{b}_{\text{goal}} - \mathbf{o}_B \,| \quad \text{and} \quad \boldsymbol{d}_{\text{goal}} = \frac{\mathbf{b}_{\text{goal}} - \mathbf{o}_B}{M_{\text{goal}}} \tag{C.1}$$

It then estimates the global probability, that there exists any action which can achieve a behaviour whose component, in the requested direction, has at least the requested magnitude.

This global probability is computed using the following approximation:

There exists an action which can attain the goal magnitude in the goal direction	$\Leftrightarrow$	There exists an extreme action which can attain at least the goal magnitude in the goal direction

(C.2)

where an extreme action is an action such that each component is either the minimum or maximum of the permitted values. If the two events of Equation C.2 are considered equivalent then their probabilities are the same. The number of extreme actions is $2^{Dim(\mathbf{Action})}$. For each of these, the probability of achieving a behaviour component at

least as large as that requested can be computed using the probability of success heuristic which was used in Section 8.1.

$$\mathbf{Prob}(B.\boldsymbol{d}_{\mathbf{goal}} > M_{\mathbf{goal}} \mid \mathbf{s}_{\mathbf{current}}, a_q, \mathbf{E}) = \mathrm{erf}\left(\frac{(\boldsymbol{d}_{\mathbf{goal}} \cdot \mathbf{b}_{\mathbf{near}}^{\mathbf{smooth}}) - M_{\mathbf{goal}}}{C \mid (\mathbf{s}_{\mathbf{current}}, a_q) - (\mathbf{s}_{\mathbf{near}}, \mathbf{a}_{\mathbf{near}}) \mid}\right) \quad \text{(C.3)}$$

where a_q is the qth extreme action and where $(\mathbf{s}_{\mathbf{near}}, \mathbf{a}_{\mathbf{near}}, \mathbf{b}_{\mathbf{near}}, \mathbf{b}_{\mathbf{near}}^{\mathbf{smooth}})$ is the nearest neighbour to $(\mathbf{s}_{\mathbf{current}}, a_q)$ in the exemplar set $\mathbf{E}$.

The probability that all of these actions are unable to achieve a large enough component is no more than the probability that the most promising cannot achieve the magnitude (from $\mathbf{Prob}(A \cap B) \leq \mathbf{Prob}(A)$). So if the probability of success of the most promising action is less than some resignation probability, $P_{\mathbf{res}}$, then this estimate recommends that the *SAB* action chooser resign itself to using the best known action.

The computational expense of this scheme is not great for a low dimensional action space ($Dim(\mathbf{Action}) \leq 4$). For a higher dimensional space it might be necessary to perform a hill-climbing search among extreme actions to find an approximation to the most promising extreme action.

Bibliography

[Aboaf *et al.*, 1989] E. W. Aboaf, S. M. Drucker, and C. G. Atkeson. Task-Level Robot Learning: Juggling a Tennis Ball More Accurately. In *IEEE International Conference on Robotics and Automation*, 1989.

[Aha *et al.*, 1990] D. W. Aha, D. Kibler, and M. K. Albert. Instance-Based Learning Algorithms. *To appear in Machine Learning*, 1990.

[Albus, 1975a] J. S. Albus. A New Approach to Manipulator Control: Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, pages 220–227, September 1975.

[Albus, 1975b] J. S. Albus. Data Storage in the Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, pages 228–233, September 1975.

[Albus, 1981] J. S. Albus. *Brains, Behaviour and Robotics.* BYTE Books, McGraw-Hill, 1981.

[An *et al.*, 1988] C. H. An, C. G. Atkeson, and J. M. Hollerbach. *Model-Based Control of a Robot Manipulator.* M. I. T. Press, 1988.

[Angus, 1989] J. E. Angus. On the connection between neural network learning and multivariate non-linear least squares estimation. *Neural Networks*, 1(1), January 1989.

[Atkeson and Reinkensmeyer, 1989] C. G. Atkeson and D. J. Reinkensmeyer. Using Associative Content-Addressable Memories to Control Robots. A. I. Memo 1124, M. I. T. Artificial Intelligence Laboratory, November 1989.

[Atkeson, 1989] C. G. Atkeson. Using Local Models to Control Movement. In *Proceedings of Neural Information Processing Systems Conference*, November 1989.

[Barto *et al.*, 1983] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike Adaptive elements that that can learn difficult Control Problems. *IEEE Transactions on Systems Man and Cybernetics*, 1983.

[Bentley, 1980] J. L. Bentley. Multidimensional Divide and Conquer. *Communications of the ACM*, 23(4):214—229, 1980.

[Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, 1984.

[Burghes and Graham, 1980] D. Burghes and A. Graham. *Introduction to Control Theory including Optimal Control.* Ellis Horwood, 1980.

[Burkhill, 1978] J. C. Burkhill. *A First Course in Mathematical Analysis.* Cambridge University Press, 1978.

[Christiansen *et al.*, 1990] A. D. Christiansen, M. T. Mason, and T. M. Mitchell. Learning Reliable Manipulation Strategies without Initial Physical Models. In *IEEE Conference on Robotics and Automation*, pages 1224–1230, 1990.

[Cleveland and Delvin, 1988] W. S. Cleveland and S. J. Delvin. Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting. *Journal of the American Statistical Association*, 83(403):596–610, September 1988.

[Cleveland, 1979] W. S. Cleveland. Robust Locally Weighted Regression and Smoothing Scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, December 1979.

[Clocksin and Moore, 1989] W. F. Clocksin and A. W. Moore. Some Experiments in Adaptive State Space Robotics. In *Proceedings of the 7th AISB Conference, Brighton.* Morgan Kaufman, April 1989.

[Connell and Utgoff, 1987] M. E. Connell and P. E. Utgoff. Learning to control a dynamic physical system. In *Proceedings of the American Association for Artificial Intelligence Conference*, 1987.

[Diederich, 1990] J. Diederich. An Explanation Component for a Connectionist Inference System. In L. C. Aiello, editor, *Proceedings of ECAI90: the 9th European Conference on Artificial Intelligence*, pages 222–227, August 1990.

[Duda and Hart, 1973] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[Farmer and Sidorowich, 1988] J. D. Farmer and J. J. Sidorowich. Predicting Chaotic Dynamics. In J. A. S. Kelso, A. J. Mandell, and M. F. Shlesinger, editors, *Dynamic Patterns in Complex Systems.* World Scientific, New Jersey, 1988.

[Firschein and others, 1986] O. Firschein et al. Teleoperation and Robotics Scenarios. In *Artificial Intelligence for Space Station Automation*, pages 288–305. NASA Advanced Technology Advisory Committee, 1986.

[Franke, 1982] R. Franke. Scattered Data Interpolation: Tests of Some Methods. *Mathematics of Computation*, 38(157), January 1982.

[Friedman *et al.*, 1977] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[Fu *et al.*, 1987] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, Sensing, Vision and Intelligence.* McGraw-Hill, 1987.

[Fu, 1970] K. S. Fu. Learning Control Systems—Review and Outlook. *IEEE Transactions on Automatic Control*, pages 210–221, April 1970.

[Gordon and Grefenstette, 1990] D. F. Gordon and J. J. Grefenstette. Explanations of Empirically Derived Reactive Plans. In *Proceedings of the 7th International Conference on Machine Learning*, June 1990.

[Gottschalk *et al.*, 1989] P. G. Gottschalk, J. L. Turney, and T. N. Mudge. Efficient Recognition of Partially Visible Objects Using a Logarithmic Complexity Matching Technique. *International Journal of Robotics Research*, 8(6), December 1989.

[Grosse, 1989] E. Grosse. LOESS: Multivariate Smoothing by Moving Least Squares. In *Approximation Theory VI.* Academic Press, 1989.

[Hearn, 1973] A. C. Hearn. REDUCE 2 User's Manual. Technical report, University of Utah, March 1973.

[Hoel, 1971] P. G. Hoel. *Introduction to Mathematical Statistics.* Wiley International, 1971.

[Holland *et al.*, 1987] J. H. Holland, L. B. Booker, and D. E. Goldberg. Classifier Systems and Genetic Algorithms. Technical Report No. 8, University of Michigan, Cognitive Science and Machine Intelligence Laboratory, 1987.

[Jordan and Jacobs, 1990] M. I. Jordan and R. A. Jacobs. Learning to Control an Unstable system with Forward Modeling. Technical report, M. I. T., 1990.

[Kaelbling, 1990a] L. P. Kaelbling. Learning Functions in k-DNF from Reinforcement. In *Proceedings of the 7th International Conference on Machine Learning*, June 1990.

[Kaelbling, 1990b] L. P. Kaelbling. Learning in Embedded Systems. PhD. Thesis. Technical Report No. TR-90-04, Stanford University, Department of Computer Science, 1990.

[Kibler *et al.*, 1988] D. Kibler, D. W. Aha, and M. K. Albert. Instance-Based Prediction of Real-Valued Attributes. Technical report 88-07, University of California, Irvine, 1988.

[Langley *et al.*, 1983] P. Langley, G. L. Bradshaw, and H. A. Simon. Rediscovering Chemistry with the BACON System. In *Machine Learning: An Artificial Intelligence Approach.* Morgan Kaufmann, 1983.

[Maclaren, 1989] N. M. Maclaren. Some Empirical Algorithms for non-Euclidean Nearest Neighbour Searching. Document at Cambridge University Computer Laboratory, December 1989.

[Mel, 1988] B. W. Mel. MURPHY: A Robot that Learns by Doing. Technical Report UIUCDCS-R-88-1397, University of Illinois at Urbana-Champaign, January 1988.

[Mel, 1989] B. W. Mel. MURPHY: A Connectionist Approach to Vision-Based Robot Motion Planning. Technical Report CCSR-89-17A, University of Illinois at Urbana-Champaign, June 1989.

[Michie and Chambers, 1968] D. Michie and R. A. Chambers. BOXES: An Experiment in Adaptive Control. In *Machine Intelligence 2.* Oliver and Boyd, 1968.

[Michie, 1989] D. Michie. Personal Models of Rationality. *Journal of Statistical Planning and Inference*, 21, 1989. Also published as a Turing Institute Technical Report.

[Miller *et al.*, 1987] W. T. Miller, F. H. Glanz, and L. G. Kraft. Application of a General Learning Algorithm to the Control of Robotic Manipulators. *International Journal of Robotics Research*, 6(2), 1987.

[Miller, 1989] W. T. Miller. Real-Time Application of Neural Networks for Sensor-Based Control of Robots with Vision. *IEEE Transactions on systems, Man and Cybernetics*, 19(4):825–831, July 1989.

[Minsky and Papert, 1969] M. Minsky and S. Papert. *Perceptrons.* M. I. T. Press, 1969.

[Moore, 1990] A. W. Moore. Acquisition of Dynamic Control Knowledge for a Robotic Manipulator. In *Proceedings of the 7th International Conference on Machine Learning*, June 1990.

[Murray, 1972] W. Murray. *Numerical Methods for Unconstrained Optimization.* Academic Press, 1972.

[Omohundro, 1987] S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. Technical Report UIUDCS-R-87-1331, University of Illinois at Urbana-Champaign, April 1987.

[Phan *et al.*, 1990] M. Phan, J. N. Juang, and R. W. Longman. Recent Developments in Learning Control and System Identification for Robots and Structures. In *Dynamics of Space structures Conference, Cranfield Institute of Technology*, June 1990.

[Poggio and Girosi, 1989] T. Poggio and F. Girosi. Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks. *Science*, 247:978–982, 1989.

[Preparata and Shamos, 1985] F. P. Preparata and M. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[Quinlan, 1983] J. R. Quinlan. Learning Efficient Classification Procedures and their Application to Chess End Games. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning—An Artificial Intelligence Approach (I)*. Tioga Publishing Company, Palo Alto, 1983.

[Raibert, 1978a] M. H. Raibert. A Model for Sensorimotor Control and Learning. *Biological Cybernetics*, 29:29–36, 1978.

[Raibert, 1978b] M. H. Raibert. Motor Control and Learning by the State Space Model. PhD. Thesis, M. I. T., 1978.

[Rendell, 1983] L. Rendell. A New Basis for State Space Learning Systems and a Successful Implementation. *Artificial Intelligence*, pages 369–392, 1983.

[Rumelhart and McClelland, 1984] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing—Explorations in the Microstructure of Cognition.* M. I. T. Press, 1984.

[Salzberg, 1988] S. Salzberg. Exemplar-based learning: theory and implementation. Technical report, Aiken Computation Laboratory, Harvard University, 1988.

[Sammut and Michie, 1989] C. Sammut and D. Michie. Controlling a 'Black Box' Simulation of a Spacecraft. Technical Report TIRM-89-039, Turing Institute, October 1989.

[Samuel, 1967] A. L. Samuel. Some Studies in Machine Learning using the Game of Checkers II—Recent Progress. Memo No. 52, Stanford Artificial Intelligence Project, June 1967.

[Simons *et al.*, 1982] J. Simons, H. Van Brussel, J. De Schutter, and J. Verhaert. A Self-Learning Automaton with Variable Resolution for High Precision Assembly by Industrial Robots. *IEEE Transactions on Automatic Control*, 27(5):1109–1113, October 1982.

[Stanfill and Waltz, 1986] C. Stanfill and D. Waltz. Towards Memory-Based Reasoning. *Communications of the ACM*, 29(12):1213–1228, December 1986.

[Sutton, 1990] R. S. Sutton. Integrated Architecture for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the 7th International Conference on Machine Learning*, June 1990.

[Utgoff, 1989] P. E. Utgoff. Incremental Induction of Decision Trees. *Machine Learning*, 4:161–186, 1989.

[Valiant, 1984] L. G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[Vogel, 1989] T. U. Vogel. PhD. Thesis Proposal. First Year Report at University of Cambridge Computer Laboratory, May 1989.

[Whitley, 1989] D. Whitley. Applying Genetic Algorithms to Neural Network Learning. In *Proceedings of the 7th AISB Conference, Brighton.* Morgan Kaufman, April 1989.

[Zrimec and Mowforth, 1990] T. Zrimec and P. Mowforth. Learning by an Autonomous Agent in the Pushing Domain. In *Machine Intelligence 12 (to appear shortly).* Oliver and Boyd, 1990.