

2818 94-36018

100

NOV 23 1994

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Brain and Cognitive Sciences and
The Artificial Intelligence Laboratory
Rm. NE43-771 • 545 Technology Square, Cambridge, MA 02139
(617) 253-0788 • cga@ai.mit.edu

August 7, 1994

Marc Jacobs
AFOSR/NM
Bldg. 410
Bolling AFB, DC 20332-6448

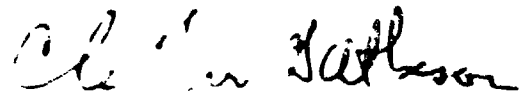
Dear Dr. Suddarth,

This is the Final Technical Report for the grant "Learning Maneuvers Using Neural Network Models" (F49620-93-1-0263) for the period 4/1/93 - 3/31/94.

The objective of this research was to explore possible roles for adaptive neural networks in learning complex maneuvers. During the year-long grant period we developed and demonstrated new and more powerful versions of memory-based learning algorithms, and also developed new reinforcement learning algorithms. This progress is described in the attached papers.

The funds provided by the Air Force had a large impact on education as well as on research, providing partial support for several graduate students and undergraduates. Ben Weintraub and Jacqueline Moore finished their Master's theses during the grant period. They developed one of the testbeds for research, a small helicopter system. This research has also been incorporated in several courses, including courses on motor learning, computational approaches to motor learning, and nonparametric regression applied to learning and optimization.

Sincerely,



Christopher G. Atkeson
Associate Professor

Robot Juggling: An Implementation of Memory-based Learning

*Stefan Schaal and Christopher G. Atkeson**

Abstract

This paper explores issues involved in implementing robot learning for a challenging dynamic task, using a case study from robot juggling. We use a memory-based local modeling approach (locally weighted regression) to represent a learned model of the task to be performed. Statistical tests are given to examine the uncertainty of a model, to optimize its prediction quality, and to deal with noisy and corrupted data. We develop an exploration algorithm that explicitly deals with prediction accuracy requirements during exploration. Using all these ingredients in combination with methods from optimal control, our robot achieves fast real-time learning of the task within 40 to 100 trials.

* Address of both authors: Massachusetts Institute of Technology, The Artificial Intelligence Laboratory & The Department of Brain and Cognitive Sciences, 545 Technology Square, Cambridge, MA 02139, USA. Email: ss-schaal@ai.mit.edu, cgs@ai.mit.edu. Support was provided by the Air Force Office of Scientific Research and by Siemens Corporation. Support for the first author was provided by the German Scholarship Foundation and the Alexander von Humboldt Foundation. Support for the second author was provided by a National Science Foundation Presidential Young Investigator Award. We thank Gideon Stein for implementing the first version of LWR on the i860 microprocessor, and Gerrie van Zyl for building the devil stick robot and implementing the first version of devil stick learning.

Introduction

Learning control means improving a motor skill by repeatedly practicing a task. There has been much progress in learning control research. But many projects test proposed algorithms only in simulation. We have found that actual implementation of learning control forces us to consider issues not adequately addressed in simulations. In this paper we describe which ingredients were needed to actually implement a learning algorithm on a robot for a complicated dynamic task.

We are exploring systems that learn by explicitly remembering their experiences in order to build models of the world. The learning community distinguishes between two different methods to represent a model, *parametric* and *nonparametric*. A *parametric* model consists of a certain mathematical function which possesses a finite set of free parameters that have to be determined to make the function fit the data. This function models *all* data simultaneously, which means that parametric models correspond to *global* function fitting. Parametric models and training methods often do not remember the data they were trained on. Standard linear regression, sigmoidal neural networks, radial basis function networks, etc., belong in this class of techniques. *Nonparametric* models also have an underlying function with a set of parameters which are to be adjusted. However, the number of the parameters can grow with the amount of data and the parameters can be recalculated whenever the model is used to generate an output from a new input (a process which is also called a lookup or query). This makes sense if not all data is taken into account to estimate the parameters but merely a subset, or if individual data points are weighted differently with respect to different query points. Common algorithms to choose the subset or the weighting are, for example, n-nearest neighbor methods (e.g., [12, 25]) or kernel regression (e.g., [19, 28, 37]). Common functions are (hyper-)planes or (hyper-)quadratic surfaces. By letting only a few data points contribute to forming the parameters, these types of nonparametric models correspond to *local* function fitting: they build a *local* model to fit a subset of data points with their function. As the word "local" implies, the model will be valid only in a restricted region. Due to the necessity of continuous recalculation of the parameters for each individual query, local nonparametric models have to memorize all data and are often called memory-based. Weighted averaging and nearest neighbor methods are presumably the best known nonparametric approaches.

We are investigating a recently developed nonparametric (memory-based) statistical technique, locally weighted regression (LWR), to model the system we are trying to control [11, 15, 16]. The LWR approach allows us to efficiently estimate local linear models for different points in the state space. LWR offers a variety of statistical tools to assess the reliability of lookups, to optimize the quality of a lookup, and also to cope with noise and corrupted data. This allows the robot to monitor its own skill level, and it provides the ba-

sis for an exploratory behavior that is almost entirely driven by the stream of incoming data from practicing the task.

Our starting point for modeling is that we assume knowledge of what constitutes a state of the system, i.e., the input/output representations, but the form of the dynamics equations of the task to be controlled is unknown. Past work tested our ideas by implementing learning for one-shot or static tasks, such as throwing a ball at a target [1], and also repetitive or dynamic tasks, such as bouncing a ball on a paddle [2] and hitting a stick back and forth (a form of juggling known as devil sticking) [40]. This as well as other experimental work (e.g., [32]) has highlighted the importance of making sure the control paradigm used is robust to uncertainty, that the robot is able to compute what is known about the task, and how well it is known, and that there is some process that generates exploration, so that models and controllers based on insufficient data are improved. All these points are addressed by the LWR learning algorithm. Using our work with the devil sticking robot as an example, this paper describes what was needed to implement real-time learning based on this algorithm.

The next section of this paper discusses a number of control approaches which make use of models and motivates the choice in our work. Locally weighted regression and some of its statistical tools are introduced afterwards. Exploration, a key feature for system identification of modeling approaches, receives attention in the fourth section where we introduce a goal-directed exploration algorithm which keeps explicit control over prediction accuracy during exploration. In the fifth section, the previously introduced methods are find application in a real-time implementation of learning how to juggle the devil stick.

Control Paradigms

Before discussing the details of our representational approach, it is useful to consider some of the alternative control paradigms that might make use of learned models.

Deadbeat Control

In considering repetitive or dynamic tasks, we will focus on nonlinear regulator design, assuming there is a desired state $\mathbf{x}_d$ ¹ to achieve. Since often the observations of system inputs and outputs occur at discrete time intervals and not all the derivatives of the state are typically measured, we restrict our analysis to discrete time models. The notation for the forward dynamics model of a discrete system is

¹ Our notation has the following conventions: scalars are denoted by lower case letters in *italic* face (e.g., s), vectors are denoted by lower case letters in **bold** face (e.g., $\mathbf{v}$), matrices are denoted by upper case letters in **bold** face (e.g., $\mathbf{M}$), scalar valued function are in *italic* face (e.g., $f()$), vector valued function are in **bold** face (e.g., $\mathbf{f}()$), and $()^T$ denotes the transpose of a vector or matrix, whereby all vectors are originally column vectors. The ^ (caret) indicates models and predictions by models. Dots on top of variables indicate time derivatives.

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{f}}(\mathbf{x}_k, \mathbf{u}_k) \quad (2.1)$$

and attempts to perform the task generate experience vectors $(\mathbf{x}_k^T, \mathbf{u}_k^T, \mathbf{x}_{k+1}^T)^T$. A straightforward approach to improving performance on the task is to learn an inverse model

$$\hat{\mathbf{u}}_k = \hat{\mathbf{f}}^{-1}(\mathbf{x}_k, \mathbf{x}_{k+1}) \quad (2.2)$$

from the database of experiences and use the model to predict commands for later attempts of the task by replacing $\mathbf{x}_{k+1}$ in (2.2) by a desired state $\mathbf{x}_{k+1, \text{desired}}$. Another approach is to learn the forward model (2.1) and then search for a good command, minimizing the (by $\mathbf{Q}$ and $\mathbf{R}$ weighted) squared magnitude of the predicted state error and the command:

$$\min_{\mathbf{u}_k} \left[(\hat{\mathbf{f}}(\mathbf{x}_k, \mathbf{u}_k) - \mathbf{x}_d)^T \mathbf{Q} (\hat{\mathbf{f}}(\mathbf{x}_k, \mathbf{u}_k) - \mathbf{x}_d) + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k \right], \quad (2.3)$$

Eq.(2.2) and Eq.(2.3) with $\mathbf{R} = \mathbf{0}$ correspond to deadbeat control.

The deadbeat controllers above did not achieve satisfying robustness in our work since they attempt to cancel the plant dynamics entirely. A less aggressive nonlinear control approach is to locally linearize the system about the desired point, and then use one of the many linear controller design techniques, e.g., pole placement, linear quadratic (LQ), or H_∞ . Such an approach is very successful if the system remains within the linear region.

Representing the Forward Model

Modeling approaches require model representations. If the nonlinear system has a particular structure, it can be globally linearized using nonlinear coordinate transformations and state feedback (feedback linearization) [36]. Any linear control design techniques may be used subsequently. Much of the recent work in adaptive controllers for nonlinear systems assumes some knowledge of the form of the nonlinearities and the plant's unknown parameters [30]. A common formulation requires the plant be representable accurately by a feedback linearizable model in which all unknown elements appear linearly as a parameter vector. A more black box approach to adaptive control [21] is to use a form of parametric Volterra series in the inputs and states. Single hidden layer perceptron-like neural network models essentially project the input data along a line given by the input weights, and then output a one dimensional function of the value of that projection. Radial basis function networks use centers of spherically symmetric contributions from parameterized one dimensional functions applied to the distance between each input and the center. All these approaches make implicit assumptions about the form of the system they are interacting with, which we want to avoid, as will be demonstrated in the next section.

Optimal Control Approaches

Learning approaches that do not commit to a particular representational form generate numerical representations, for which optimal control techniques provide natural methods to

design control systems for nonlinear tasks. Dynamic programming [8, 9, 13] lays the basis for a general paradigm of nonlinear controllers. In our formulation of the regulation problem, a goal state $\mathbf{x}_d$ is given, which is typically an equilibrium state, so $\mathbf{x}_d = \mathbf{f}(\mathbf{x}_d, 0)$. A one step cost $L(\mathbf{x}, \mathbf{u})$ is defined over all states and controls. The criterion to be optimized is the infinite horizon sum of one step costs starting at the current time:

$$J = \sum_{k=1}^{\infty} L(\mathbf{x}_k, \mathbf{u}_k). \quad (2.4)$$

We typically require either a temporal discount factor or $L(\mathbf{x}_d, 0) = 0$ to ensure well defined solutions to the optimization problem. The value function $V(\mathbf{x})$ is the optimal cost created by solving (2.4) starting in state $\mathbf{x}$. At any point, a globally optimal control action can be chosen by the nonlinear controller by solving the local optimization problem:

$$\mathbf{u}^{opt} = \arg \min_{\mathbf{u}} [L(\mathbf{x}, \mathbf{u}) + V(f(\mathbf{x}, \mathbf{u}))]. \quad (2.5)$$

If one assumes a locally linear model of the plant,

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{f}}(\mathbf{x}_k, \mathbf{u}_k) \approx \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{c}, \quad (2.6)$$

a weighted locally quadratic model of the one step cost,

$$L(\mathbf{x}, \mathbf{u}) \approx \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} + \frac{1}{2} \mathbf{u}^T \mathbf{R} \mathbf{u} + \mathbf{x}^T \mathbf{S} \mathbf{u} + \mathbf{t}^T \mathbf{u}, \quad (2.7)$$

and a locally quadratic model of the value function,

$$V(\mathbf{x}) \approx V_0 + \mathbf{V}_1^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{V}_2 \mathbf{x}, \quad (2.8)$$

one can compute a locally optimal command analytically:

$$\mathbf{u}^{opt} = -(\mathbf{R} + \mathbf{B}^T \mathbf{V}_2 \mathbf{B})^{-1} (\mathbf{B}^T \mathbf{V}_2 \mathbf{A} \mathbf{x} + \mathbf{S}^T \mathbf{x} + \mathbf{B}^T \mathbf{V}_2 \mathbf{c} + \mathbf{V}_1^T \mathbf{B} + \mathbf{t}). \quad (2.9)$$

Unfortunately, value functions are difficult to represent and to compute, even though this can be done off-line. Predictive control design techniques avoid using a value function, but are then merely locally optimal [10]. Value functions can also be approximated, e.g., with neural networks [42]. We are interested in exploring approximations to value functions that produce a locally quadratic model of $V(\mathbf{x})$ in a local neighborhood of $\mathbf{x}$.

In this paper we are working within an optimal control framework. We would like to design a fully nonlinear controller from a full computation of the optimal value function. This is currently too expensive to compute, so we use linear quadratic (LQ) regulator techniques to approximate the value function and design a corresponding controller. We make extensive use of local linear models of the system to be controlled. The linearized models are calculated on an as needed basis and are recalculated with each new piece of data to update the controller. All of this happens in real time as the robot is executing the task.

Locally Weighted Regression

The point of view explored in this paper is that the goal of a learning system for robots is to be able to build internal models of tasks during execution of those tasks. These models are multidimensional functions that are approximated from sampled data (the previous experiences or attempts to perform the task). The learned models are used in a variety of ways to successfully execute the task. We would like the models to incorporate the latest information. The models will be continuously updated with a stream of new training data, so updating a model with new data should take a short period of time. There are also time constraints on how long it can take to use a model to make a prediction. Because we are interested in control methods that make use of local linearizations of the plant model, we want a representation that can quickly compute a local linear model of the represented transformation. We would also like to minimize the negative interference from learning new knowledge on previously stored information.

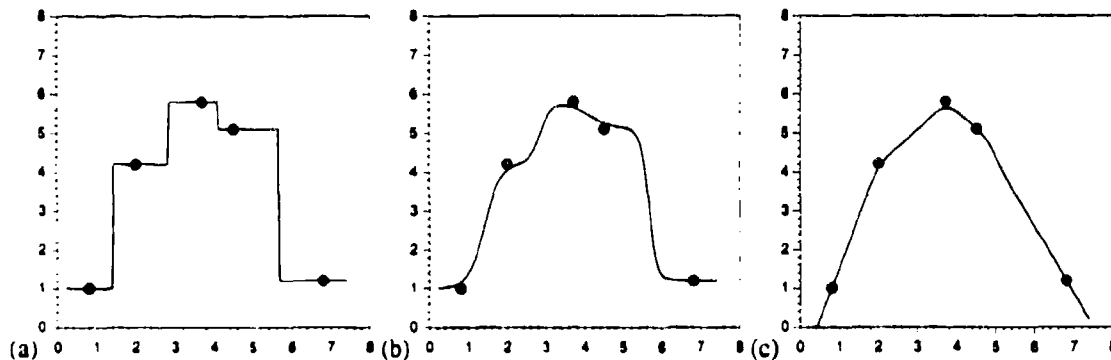


Figure 1: Characteristic performance of three different nonparametric function approximation techniques: (a) nearest neighbor; (b) weighted average; (c) locally weighted regression

As the most generic approximator that satisfies many of these criteria, we explore a version of memory-based learning techniques called locally weighted regression (LWR). [15, 16, 11, 6, 24, 14, 27]. A memory-based learning (MBL) system is trained by storing the training data in a memory. This allows MBL systems to achieve real-time learning. MBL avoids interference between new and old data by retaining and using all the data to answer each query. MBL approximates complex functions using simple local models, as does a Taylor series. Examples of types of local models include nearest neighbor, weighted average, and locally weighted regression. Each of these local models combine points near to a query point to estimate the appropriate output. Figure 1 shows typical curve fits for each of these methods.

Locally weighted regression uses a relatively complex regression procedure to form the local model, and is thus more expensive than nearest neighbor and weighted average

memory-based learning procedures. For each query a new local model is formed. The rate at which local models can be formed and evaluated limits the rate at which queries can be answered. This paper describes how locally weighted regression can be implemented in real time.

An unweighted regression finds the solution to the equations:

$$\mathbf{y} = \mathbf{X} \cdot \boldsymbol{\beta} \quad (3.1a)$$

by solving the normal equations:

$$\mathbf{X}^T \mathbf{X} \boldsymbol{\beta} = \mathbf{X}^T \mathbf{y}, \quad (3.1b)$$

where $\mathbf{X}$ is an $m \times (n+1)$ matrix consisting of m data points, each represented by its n input dimensions and a "1" in the last column, $\mathbf{y}$ is a vector of corresponding outputs for each data point, $\boldsymbol{\beta}$ is the $n+1$ vector of unknown regression parameters, and J is the sum of squared errors over all given data points (cf. Table 1, Appendix A). Solving for $\boldsymbol{\beta}$ yields

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}, \quad (3.2)$$

and a prediction of the outcome of a query point $\mathbf{x}_q$ becomes:

$$\hat{y}_q = \mathbf{x}_q^T \boldsymbol{\beta}. \quad (3.3)$$

However, this gives distant points equal influence with nearby points on the ultimate answer to the query, for equally spaced data. To weight similar points more, locally weighted regression is used. First, a distance is calculated from each of the stored data points (rows in the $\mathbf{X}$ matrix) to the query point $\mathbf{x}_q$:

$$d_i^2 = \sum_{j=1}^n s_j (\mathbf{X}_{ij} - \mathbf{x}_{qj})^2. \quad (3.4)$$

The factor s_j reflects a positive weighting (distance metric) among the n input dimensions, either to normalize those or to give them different importance. The weight for each stored data point is a function of the distance (3.4):

$$w_i = f(d_i^2). \quad (3.5)$$

Each row i of $\mathbf{X}$ and $\mathbf{y}$ is multiplied by the corresponding weight w_i . A simple weighting function just raises the distance (3.4) to a negative power, which determines how local the regression will be (the rate of drop-off of the weights with distance):

$$w_i = \frac{1}{d_i^k}. \quad (3.6)$$

This type of weighting function goes to infinity as the query point approaches a stored data point which forces the locally weighted regression to exactly match that stored point. If the data is noisy, exact interpolation is not desirable, and a weighting scheme with limited

magnitude is more appropriate. One such scheme, which we use in what follows, is a Gaussian kernel:

$$w_i = \exp\left(\frac{-d_i^2}{2k^2}\right). \quad (3.7)$$

The parameter k scales the size of the kernel to determine how local the regression will be. Such a weighting is used in Figure 1b and Figure 1c.

A potential problem is that the data points may be distributed in such a way as to make the regression matrix $\mathbf{X}$ singular. Ridge regression is used to prevent problems due to a singular data matrix. The following equation, with $\mathbf{X}$ and $\mathbf{y}$ already weighted, is solved for β :

$$(\mathbf{X}^T \mathbf{X} + \Lambda) \beta = \mathbf{X}^T \mathbf{y}, \quad (3.8)$$

where Λ is a diagonal matrix with small positive diagonal elements λ_i^2 . Ridge regression is equivalent to adding fake data in each direction that has a small weight and a zero output value. The ridge regression constants can also be thought of as Bayesian priors on the variance of the estimated parameter vector β .

Assessing the computational cost

A lookup in a LWR model has three stages: forming weights, forming the regression matrix, and solving the normal equations. Let us examine how the cost of each of these stages grows with the size of the data set and dimensionality of the problem. We will assume a linear local model.

Forming and applying the weights involves scanning the entire data set, so it scales linearly with the number of data points in the database m . For each of n input dimensions there are a constant number of operations, so the number of operations scales linearly with the number of input dimensions. Note that we can eliminate points whose distance exceeds a threshold, reducing the number of points considered in subsequent computational stages.

Each element of $\mathbf{X}^T \mathbf{X}$ and $\mathbf{X}^T \mathbf{y}$ is the inner (dot) product of two columns of $\mathbf{X}$ or $\mathbf{y}$. The architecture of digital signal processors is ideally suited for this computation, which consists of repeated multiplies and accumulates. The computation is linear in the number of rows m and quadratic in the number of columns $(n^2 + n*o)$, where o is the number of output dimensions.

Solving the normal equations is done using a LDL^T decomposition, which is cubic in the number of input dimensions, and independent of the number of data points. Other more sophisticated and more expensive decompositions, such as the singular value decomposition, are unnecessary since the ridge regression procedure guarantees well-conditioned normal equations.

The most straightforward parallel implementation of LWR would distribute the data points among several processors. Queries can be broadcast to the processors, and each processor can weight its data set and form its contribution to $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{y}$. These contributions can be summed and the full normal equations solved on a single processor. The communication costs are linear in the number of processors, quadratic in the number of columns ($n^2 + n*o$), and independent of the total number of points.

We have implemented the local weighted regression procedure on a 33MHz Intel i860 microprocessor. The peak computation rate of this processor is 66 MFlops. We have achieved effective computation rates of 20 MFlops on a learning problem with $n = 10$ input dimensions and $o = 5$ output dimensions, using a linear local model. This leads to a lookup time of approximately 15 milliseconds on a database of $m = 1000$ points.

Tuning The Fit Parameters

In the past we have used off-line global cross validation ([41]) to estimate reasonable values for the fit parameters: the distance metric s_j , the parameters that define the weighting function $w_i = f(d_i^2)$, and the ridge regression parameters λ_j . Since we are using a local model, that is linear in the unknown parameters, we can compute derivatives of the cross validation error $e_i = \hat{y}_i - y_i$ with respect to the fit parameters:

$$\frac{\partial e_i}{\partial s_j}, \frac{\partial e_i}{\partial k}, \frac{\partial e_i}{\partial \lambda_j},$$

and minimize the sum of the squared cross validation error using a Levenberg-Marquardt (nonlinear least squares) procedure (MINPACK, NL2SOL).

However, it is clear that these parameters should depend on the location of the query point. In this section we describe new procedures that estimate local values of the fit parameters optimized for the site of the current query point. We want to demonstrate the differences between local and global fitting in an example where we only focus on the kernel width k of a Gaussian weighting function (3.7). In Figure 2a, a noisy data set of the function $y = x - \sin^3(2\pi x^3) \cos(2\pi x^3) \exp(x^4)$ was fitted by locally weighted regression with a globally optimized, i.e. constant, k . In the left half of the plot, the regression starts to fit noise because k had to be rather small to fit the high frequency regions on the right half of the plot. The prediction intervals, which will be introduced below, indicate high uncertainty in several places. To avoid such undesirable behavior, a local optimization criterion is needed. Standard linear regression analysis provides a series of well-defined statistical tools to assess the quality of fits, such as coefficients of determination, t-tests, F-test, the PRESS-statistic, Mallows's C_p -test ([23]), confidence intervals, prediction intervals, and many more (e.g., [29]). These tools can be adapted to locally weighted regression. We do not want to discuss all possible available statistics here but rather focus on two that have proved to be useful.

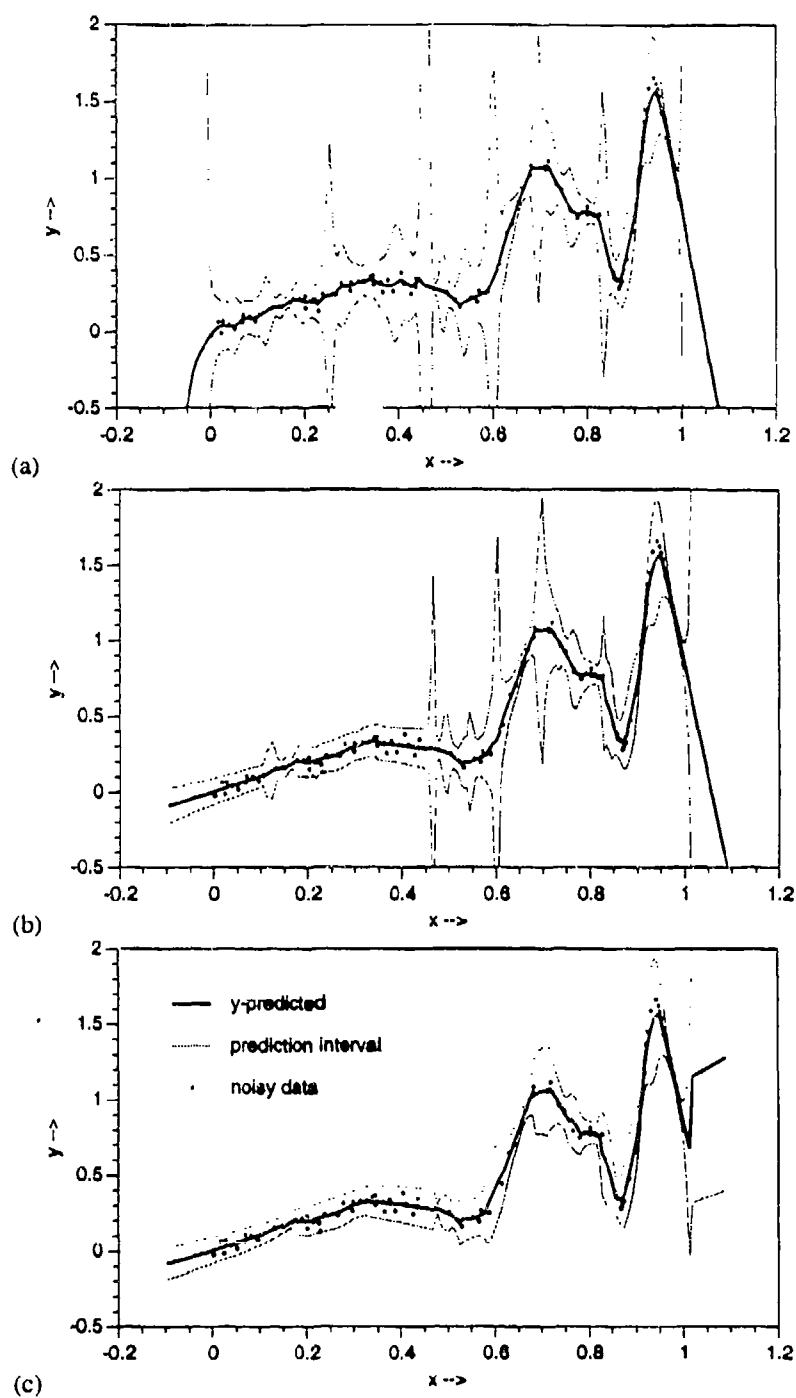


Figure 2 Optimizing the LWR fit using different measures: (a) global cross validation, (b) local cross validation, (c) local prediction intervals

Cross validation has a relative in linear regression analysis, the PRESS residual error. The PRESS statistic performs leave-one-out cross validation computationally very efficient by not requiring recalculation of the regression parameters for every excluded point. Table 1 in Appendix A shows how the PRESS residual can be expressed as a mean squared cross validation error MSE_{cross} . In Figure 2b, the same data as in Figure 2a was fitted by adjusting k to minimize MSE_{cross} at each query point. The outcome is much smoother than that of global cross validation, and also the prediction intervals are narrower. It should be noted that the extrapolation properties on both sides of the graph are quite appropriate (compared to the known underlying function), in comparison to Figure 2a and Figure 2c.

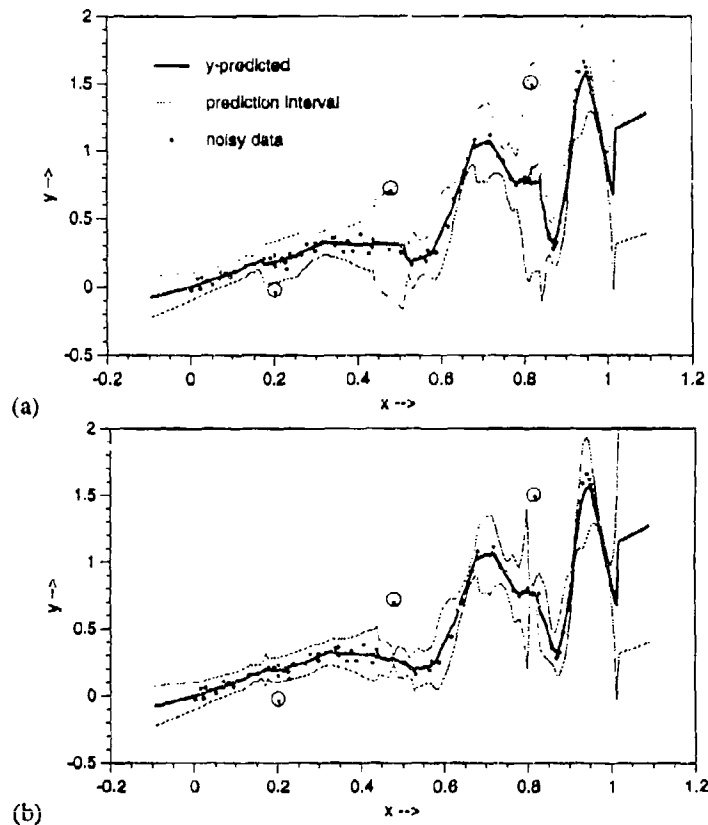


Figure 3: Influence of outliers on LWR: (a) no outlier removal, (b) with outlier removed

Prediction intervals I_q are expected bounds of the prediction error at a query point x_q . Table 1 gives the appropriate definition for LWR; its derivation can be found in most text books on regression analysis (e.g., [29]). Besides using the intervals to assess the confidence in the fit at a certain point, they provide another optimization measure. Figure 2c demonstrates the result when applying this statistic for optimizing k at each query point.

Again, the fitted curve is significantly smoother than the global cross validation fit. A rather interesting and also typical effect happens at the very right end of the plot. When starting to extrapolate, the prediction intervals suddenly favor a global regression instead of the local regression, i.e., the k was chosen to be rather large. It turns out that in local optimization one always finds a competition between local and global regression. But sudden jumps from one mode into the other take place only when the prediction intervals are so large that the data is not trustworthy anyway.

Assessing The Quality of the Local Model

Both the local cross validation error MSE_{cross} and the prediction interval I_q may serve to assess the quality of the local fit:

$$Q_{fit} = \frac{\sqrt{MSE_{cross}}}{c} \quad \text{or} \quad Q_{fit} = \frac{I_q^+ - I_q^-}{c}.$$

The factor c makes Q_{fit} dimensionless and normalizes it with respect to some user defined quantity. In our applications, we usually preferred Q_{fit} based on the prediction intervals, which is the more conservative assessment.

Dealing with Outliers

Linear regression analysis is not robust with respect to outliers. This also holds for locally weighted regression, although the influence of outliers will not be noticed unless the outliers lie close enough to a query point. In Figure 3a we added three outliers to the test data of Figure 2 to demonstrate this effect; the charts in Figure 2 should be compared to Figure 2c. [27] applied the *median absolute deviation* procedure from robust statistics [18] to globally remove outliers in LWR. We would like to localize our criterion for outlier removal. The PRESS statistic can be modified to serve as an outlier detector in LWR. For this, we need the standardized individual PRESS residual $e_{i,cross}$ (see Table 1, Appendix A). This measure has zero mean and unit variance. If, for a given data point x_i , it deviates from zero more than a certain threshold, the point can be called an outlier. A conservative threshold would be 1.96, discarding all points lying outside the 95% area of the normal distribution. In our applications, we used 2.57 cutting off all data outside the 99% area of the normal distribution. As can be seen in Figure 3b, the effects of outliers is reduced.

The Shifting Setpoint Exploration Algorithm

Learning algorithms which assume no a priori structure of the world often face the problem of sparse data in high dimensional spaces. Random exploration in order to build models of such worlds will take a very long time. Random exploration in an unknown world may also cause the system to enter unsafe or costly regions of operation. We want to develop an exploration algorithm which explicitly deals with such problems.

The shifting setpoint algorithm (SSA) attempts to decompose the control problem into two separate control tasks on different time scales. At the fast time scale, it acts as a nonlinear regulator by trying to keep the controlled system at some chosen setpoints. On a slower time scale, the setpoints are shifted to accomplish a desired goal. The SSA tries to explore the world by going to the fringes of its data support in the direction of the goal. It sets the setpoints in the fringes until statistically sufficient data has been collected to make a further step towards the goal. In this way the SSA builds a narrow tube of data support in which it knows the world. This data can be used by more sophisticated control algorithms for planning or further exploration.

We want to graphically illustrate the algorithm in a simple example of a mountain car (Figure 4) [26]. The task of the car is to drive at a given constant *horizontal* speed $\dot{x}_{desired}$ from the left to the right of the picture. $\dot{x}_{desired}$ need not be met precisely; the car should also minimize its fuel consumption. Initially, the car knows nothing about the world and cannot look ahead, but it has noisy feedback of its position and velocity. Commands, which correspond to the thrust F of the motor, can be generated at 5Hz.

The mountain car starts at its start point with one arbitrary initial action for the first time step; then it brakes and starts all over again, assuming the system can be reset somehow. The discrete one step dynamics of the car are modeled by an LWR forward model:

$$\hat{\mathbf{x}}_{next} = \hat{\mathbf{f}}(\mathbf{x}_{current}, F), \text{ where } \mathbf{x} = (\dot{x}, x)^T. \quad (4.1)$$

After a few trials, the SSA searches the data in memory for the point $(\mathbf{x}_{current}^T, F, \mathbf{x}_{next}^T)_{best}^T$ whose outcome $\hat{\mathbf{x}}_{next}$ can be predicted with the smallest local confidence interval. Note that this does not imply that $\|\mathbf{x}_{next} - \hat{\mathbf{x}}_{next}\|$ is the smallest since we have noise in the data. This best point is declared the setpoint of this stage:

$$(\mathbf{x}_{S,in}^T, F_S, \mathbf{x}_{S,out}^T)^T = (\mathbf{x}_{current}^T, F, \hat{\mathbf{x}}_{next}^T)_{best}^T, \quad (4.2)$$

and its local linear model results from a corresponding LWR lookup:

$$\mathbf{x}_{S,out} = \hat{\mathbf{f}}(\mathbf{x}_{S,in}, F_S) \approx \mathbf{A}\mathbf{x}_{S,in} + \mathbf{B}F_S + \mathbf{c}. \quad (4.3)$$

Based on this linear model, an optimal LQ controller (e.g., [13]) can be constructed by minimizing the cost:

$$J = \sum_{k=1}^{\infty} \left((\mathbf{x}_k - \mathbf{x}_{S,in})^T \mathbf{Q} (\mathbf{x}_k - \mathbf{x}_{S,in}) + r(F_k - F_S)^2 \right) \quad (4.4)$$

of the regulator problem:

$$\mathbf{x}_{k+1} - \mathbf{x}_{S,out} = \mathbf{A}(\mathbf{x}_k - \mathbf{x}_{S,in}) + \mathbf{B}(F_k - F_S), \quad (4.5)$$

where $\mathbf{Q}$ and r are weight factors in matrix or scalar form, respectively. Solving this problem results in the control law:

$$F^* = -K(x_{current} - x_{setpoint}) + F, \quad (4.6)$$

F^* is the optimal command under the cost J to go from the current state $x_{current}$ to the setpoint $x_{setpoint}$ at this stage. This does not mean that the mountain car will actually reach $x_{setpoint}$ after applying F^* ; the optimal control framework only guarantees a step towards the goal which reduces the magnitude of the value function. In the given problem it will trade speed accuracy for fuel consumption; the compromise between the two factors is reflected in the choice of Q and r . After these calculations, the mountain car learned one controlled action for the first time step. However, since the initial action was chosen arbitrarily, $x_{setpoint}$ will be significantly away from the desired speed $x_{desired}$. A reduction of this error is achieved as follows. First, the SSA repeats to do one step actions with the LQ controller (which is updated with every new data point) until sufficient data was collected to reduce the size of the prediction intervals of LWR lookups for $(x_{s,in}^T, F_s)^T$ (4.3) below a certain threshold. Then it shifts the setpoint towards the goal according to the procedure:

- 1) calculate the error of the predicted output state: $err_{s,out} = x_{desired} - x_{s,out}$
- 2) take the derivative of the error with respect to the command F_s from a LWR lookup for $(x_{s,in}^T, F_s)^T$ (cf. 4.3):

$$\frac{\partial err_{s,out}}{\partial F_s} = \frac{\partial err_{s,out}}{\partial x_{s,out}} \frac{\partial x_{s,out}}{\partial F_s} = -\frac{\partial x_{s,out}}{\partial F_s} = -B, \quad (4.7)$$

and calculate a correction ΔF_s from solving:

$$-B\Delta F_s = \alpha err_{s,out}, \quad (4.8)$$

e.g., by singular value decomposition [31]; $\alpha \in [0,1]$ determines how much of the error should be compensated for in one step.

- 3) update F_s : $F_s = F_s - \Delta F_s$ and calculate the new $x_{s,out}$ with LWR (4.3).
- 4) assess the fit for the updated setpoint with prediction intervals. If the quality is above a certain threshold, continue with 1), otherwise terminate shifting.

In this way, the output state of the setpoint shifts towards the goal until the data support falls below a threshold. Now the mountain car performs several new trials with the new setpoint and the correspondingly updated LQ controller. After the quality of fit statistics rise above a threshold, the setpoint can be shifted again. As soon as the first stage's setpoint reduces the error $x_{desired} - x_{s,out}$ to become close enough to zero, a new stage is created and the mountain car tries to move one step further in its world. The entire procedure is repeated for each new stage until the car knows how to move across the landscape along its line of setpoints with the associated LQ controllers. Figure 4b and Figure 4c show the thin band of data which the algorithm collected in state space and position-action space. These two pictures together form a narrow tube of knowledge in the input space of the forward model. The car never tried more than one exploration step into unknown territory and thus increased its probability of being safe to a high level.

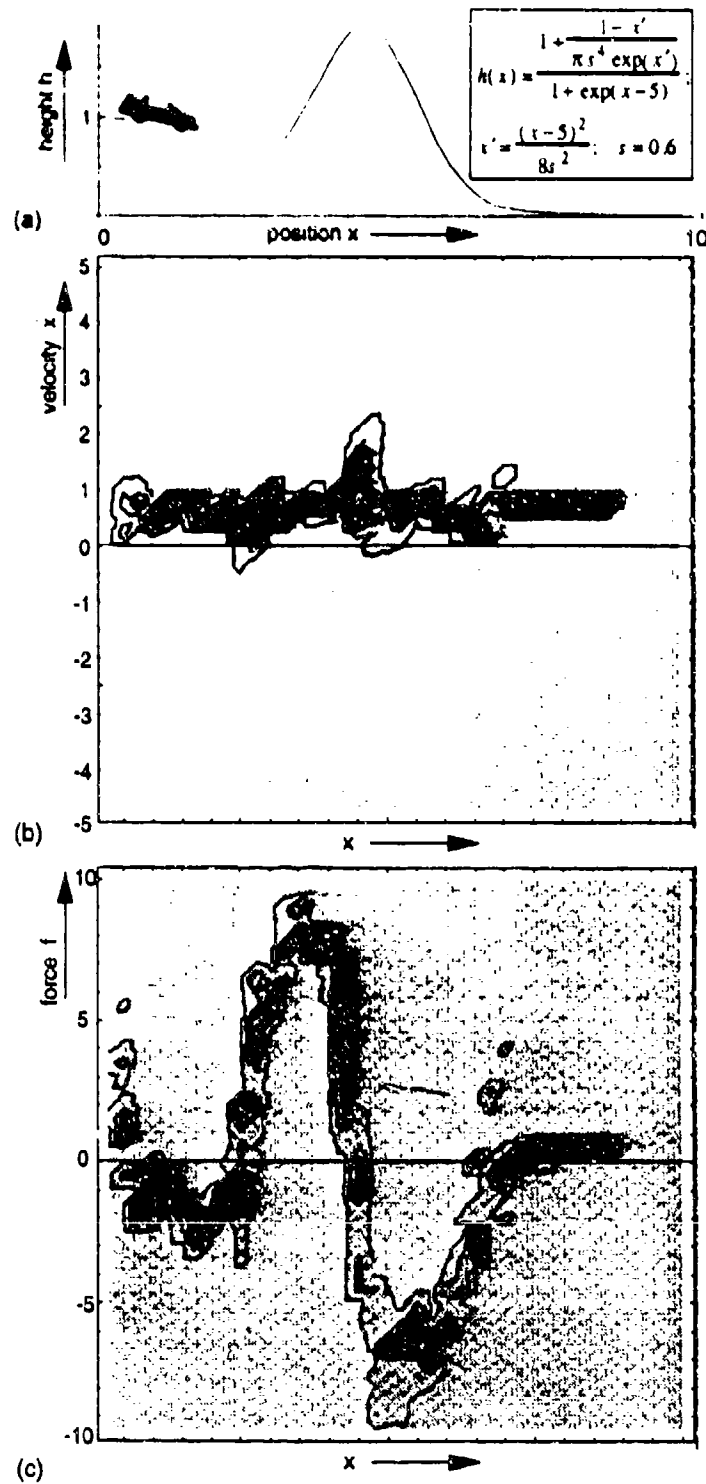


Figure 4: The mountain car: (a) landscape across which the car has to drive at constant velocity of 0.8 m/s, (b) contour plot of data density in phase space as generated by using multistage SSA, (c) contour plot of data density in position-action space

During the times where the setpoint statistics indicate insufficient data support to continue shifting, data collection is left to the randomness of the task dynamics. Thus the reduction of parameter uncertainty of the setpoint's local model also depends on this stochastic process. In order to identify the local model correctly, the stochastic process must provide data in all dimensions of the input and output space. If not, the regression problem (3.1a) may be ill-conditioned, resulting in bad estimates of the local model. Such situations were addressed by Fel'dbaum [17] as the dual control problem. In his formulations, the optimal command tries to minimize the cost and the uncertainty at the same time. So far, only expensive numerical solutions based on dynamic programming have been found to this problem [4, 7]. As an inelegant but effective way out, we add some small amount of random noise to the command F^* . The next section will demonstrate the importance of this measure.

Exploration has many facets. Depending on the task to be solved, random exploration, exploration towards unknown state space regions, and exploration towards reduction of uncertainty, etc., have been suggested [39]. The SSA exploration algorithm is goal directed and uncertainty driven under the premise not to dare any aggressive exploration outside the current data support. It is targeted at working in high dimensional environments where aggressive exploration would spend too much time in inappropriate and possibly dangerous regions. It is well suited for a real machine for which experimentation is time-consuming. The SSA requires the existence of explicit goals. However, it is not always necessary to know these goals in advance but rather let the goals develop out of the task definition, as will be shown in the next section. The SSA should be generally applicable to problems which allow a decomposition in a static exploitation and a slowly moving exploration time scale, which have one time differentiable forward dynamics, and where the noise does not exceed the capabilities of the LQ controllers.

A System For Learning Experiments: Robot Juggling

We have constructed a system for experiments in real-time motor learning [40]. The task is a juggling task known as "devil sticking". A center stick is batted back and forth between two handsticks (Figure 5a). Figures 5b,c show a sketch and photograph of our devil sticking robot. The juggling robot uses motor 1 and motor 2 to perform planar devil sticking. Hand sticks with springs and dampers are mounted on the robot to implement a passive catch: the center stick does not bounce when it hits the hand stick and requires an active throwing motion by the robot. For the time being, the problem is simplified by the center stick being constrained by a boom to move on the surface of a sphere (Figure 5b), and motor 3 is not used. For moderate amplitudes these movements are approximately planar. The boom also provides a way to measure the current state of the center stick. The task state is the predicted location at which the ballistic flight of the center stick intersects with the

hand stick held in an arbitrary but fixed nominal position $(x_{h,nominal}, y_{h,nominal})^T$. We chose $(x_{h,nominal}, y_{h,nominal})^T$ to be the hand stick position of the "upright" robot as shown in Figure 5b. As soon as the center stick does not touch the throwing hand stick anymore, standard ballistics equations for the flight of the center stick are used to map flight trajectory measurements $(x(t), y(t), \theta(t))$ into the 5-dimensional estimated task state vector, i.e., the impact state with the other hand stick held at $(x_{h,nominal}, y_{h,nominal})^T$:

$$\mathbf{x} = (p, \theta, \dot{x}, \dot{y}, \dot{\theta})^T. \quad (5.1)$$

p is the distance of the devil stick's center or mass to the impact point hand stick-devil stick (Figure 5b). The task command is given by a displacement $(x_h, y_h)^T$ of the hand stick from the nominal position $(x_{h,nominal}, y_{h,nominal})^T$, a center stick angular velocity threshold to trigger the start of a throwing motion $\dot{\theta}_t$, and a throw velocity vector $(v_x, v_y)^T$ of the hand stick, measured at point where the hand stick is attached to the robot.

$$\mathbf{u} = (x_h, y_h, \dot{\theta}_t, v_x, v_y)^T. \quad (5.2)$$

The dynamics of throwing the devilstick are thus parameterized by 5 state and 5 task commands, resulting in a 10/5-dimensional input/output model for each hand. Every time the robot catches and throws the devil stick it generates an experience vector of the form:

$$(\mathbf{x}_k^T, \mathbf{u}_k^T, \mathbf{x}_{k+1}^T)^T, \quad (5.3)$$

where $\mathbf{x}_k$ is the current state, $\mathbf{u}_k$ is the action performed by the robot, and $\mathbf{x}_{k+1}$ is the state of the center stick that results. Initially we explored learning an inverse model of the task, using nonlinear "deadbeat" control to eliminate all error on each hit. Each hand had its own inverse model of the form:

$$\hat{\mathbf{u}}_k = \hat{\mathbf{f}}^{-1}(\mathbf{x}_k, \mathbf{x}_{k+1}). \quad (5.4)$$

Before each hit, the system looked up a command with the expected impact state of the devilstick and the desired state:

$$\hat{\mathbf{u}}_k = \hat{\mathbf{f}}^{-1}(\mathbf{x}_k, \mathbf{x}_d). \quad (5.5)$$

Inverse model learning was successfully used to train the system to perform the devil stick-throwing task. Juggling runs up to 100 hits were achieved. The system incorporated new data in real time, and used databases of several hundred hits. Lookups took less than 15 milliseconds, and therefore several lookups could be performed before the end of the flight of the center stick. Later queries incorporated more measurements of the flight of the center stick and therefore more accurate predictions of the state $\mathbf{x}_k$ of the task. However, the system required substantial structure in the initial training to achieve this performance. The system was started with a default command that was appropriate for open loop performance of the task. Each control parameter was varied systematically to explore the space near the de-

fault command. A global linear model was made of this initial data, and a linear controller based on this model was used to generate an initial training set for the memory-based system (approximately 100 hits). Learning with no initial data was not possible.

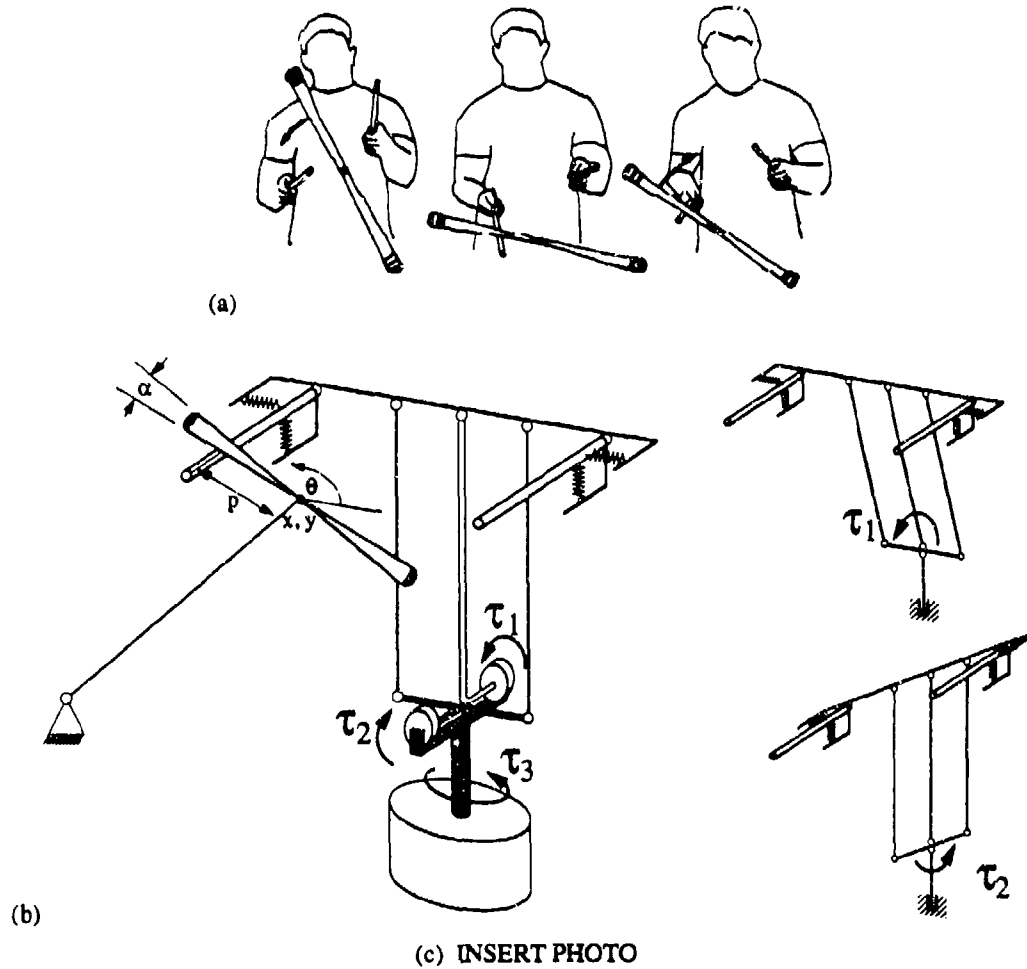


Figure 5: (a) an illustration of devil sticking, (b) sketch of our devil sticking robot: the flow of force from each motor into the robot is indicated by different shadings of the robot links, and a position change due to an application of motor 1 or motor 2, respectively, is indicated in the small sketches; (c) photograph of robot

We also experimented with learning based on both inverse and forward models. After a command is generated by the inverse model, it can be evaluated using a memory-based forward model with the same data:

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{f}}(\mathbf{x}_k, \hat{\mathbf{u}}_k). \quad (5.6)$$

Because it produces a local linear model, the LWR procedure generates estimates of the derivatives of the forward model with respect to the commands as part of the estimated parameter vector β (analog to 2.18 or 4.3). These derivatives can be used to find a correction to the command vector that reduces errors in the predicted outcome based on the forward model:

$$\frac{\partial \hat{x}}{\partial u} \Delta \hat{u} = \hat{x}_{k+1} - x_d \quad (5.7)$$

where the goal state x_d was calculated off-line from a comparison with human juggling. The pseudo-inverse of the matrix $\partial \hat{x} / \partial u$ is used to solve the above equation for $\Delta \hat{u}_k$ in order to handle situations in which the matrix is singular or a different number of commands and states exists (which does not apply for devil sticking). The process of command refinement can be repeated until the forward model no longer produces accurate predictions of the outcome. This will happen when the query to the forward model requires significant extrapolation from the current database.

We investigated this method for incremental learning of devil sticking in simulations whose dynamics were adopted from the real machine. The outcome, however, did not meet expectations: without sufficient initial data around the setpoint, the algorithm did not work. Two main reasons can be held responsible:

- i) Similar to the pure inverse model approach, the inverse-forward model acts as a one-step deadbeat controller. One-step deadbeat control applies large commands to correct for deviations from the setpoint. In the presence of errors in the model, this is detrimental since it magnifies the model errors. Additionally, the workspace bounds and command bounds of our devil sticking robot limit the size of the commands.
- ii) Due to the nonlinearities in the dynamics of the robot, the 10-dimensional input space of the forward model suffers from the first symptoms of Bellman's "curse of dimensionality". Error reduction as described in (5.7) only works if sufficient data exists at the query sites. The inevitable model errors will make the robot explore randomly, leading to dispersed data, giving little chance for model improvements. Imagine we had to place data in a (hyper-)cube of normalized edge length 0.1. A 3-dimensional input space has 10^3 such cubes leaving some probability to finally arrive at the goal. A 10-dimensional state space, however, has 10^{10} such cubes – a prohibitive number for random exploration.

Thus, two ingredients had to be added to the devil sticking controller:

- a) Control must start as soon as possible with the primary goal to increase the data density in the current region of the state-action space, and the secondary goal to arrive at the desired goal state.

- b) Control actions must avoid deadbeat properties and must be planned to go to the goal in multiple steps.

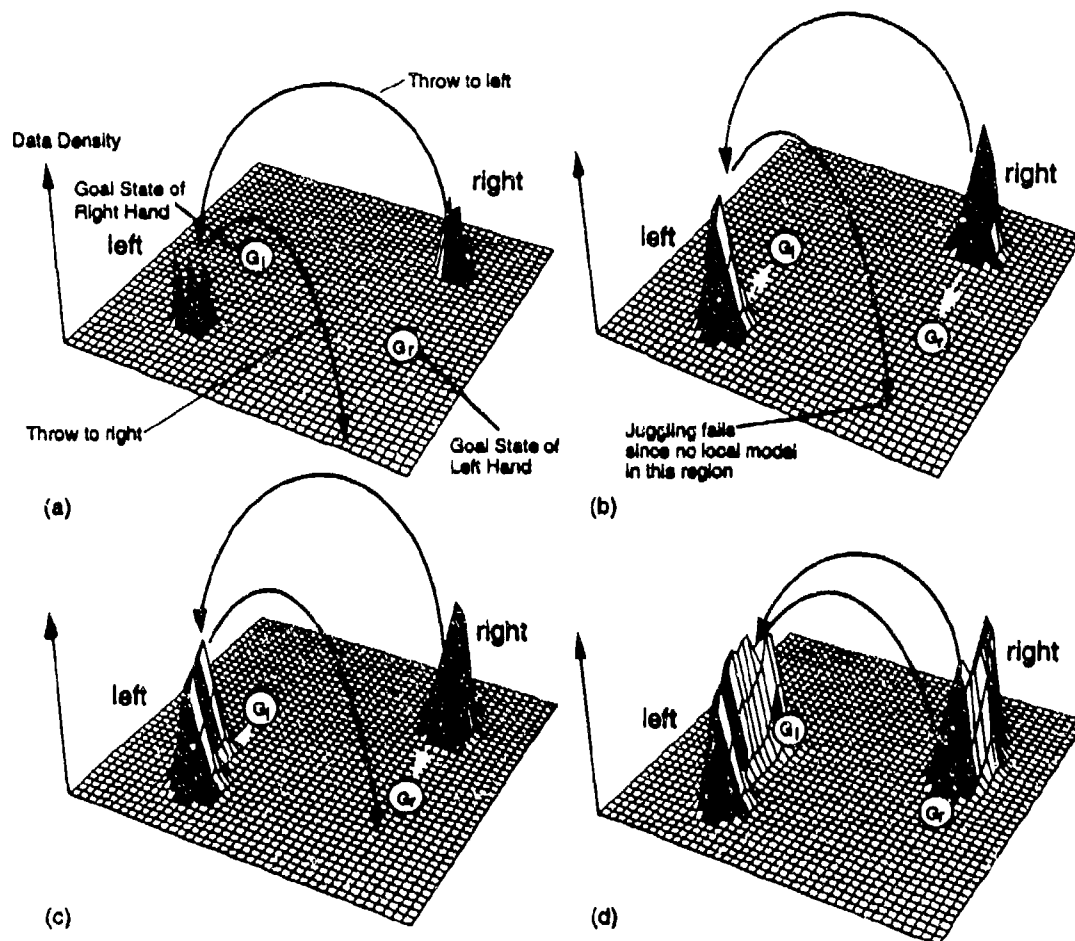


Figure 6: Abstract illustration how the SSA algorithm collects data in space: a) sparse data after the first few hits; b) high local data density due to local control in this region; c) increased data density on the way to the goals due to shifting of the setpoints; d) ridge of data density after the goal was reached

Both requirements are fulfilled by the shifting setpoint algorithm (SSA). Applied to devil sticking, the SSA proceeds as follows:

- (1) Regardless of the poor juggling quality of the robot (i.e., at most two or three hits per trial), the SSA makes the robot repeat these initial actions with small random perturbations until a cloud of data was collected somewhere in state-action space of each hand. An abstract illustration for this is given in Figure 6a to 6b.

- (2) Each point in the data cloud of each hand is used as a candidate for a setpoint of the corresponding hand by trying to predict its output from its input with LWR. The point achieving the narrowest local confidence interval becomes the setpoint of the hand and an LQ controller is calculated for its local linear model. By means of these controllers, the amount of data around the setpoints can quickly and rather accurately be increased until the quality of the local models exceeds a certain statistical threshold.
- (3) At this point, the setpoints are gradually shifted towards the goal setpoints until the data support of the local models falls below a statistical value. Shifting occurs for both input state and output state of the setpoints (cf. Eq.4.2). After shifting, the kernel k (cf. Eq. (3.7)) is optimized by minimizing the local cross validation error MSE_{cross} . (In Figure 6, the goal setpoints are given explicitly, but they actually develop automatically from the requirement to throw the devilstick increasingly close to a place, in which the other hand has data support, i.e., $x_{S,in,desired,left} = x_{S,out,right}$, $x_{S,out,desired,left} = x_{S,in,right}$, and vice versa for the other hand)
- (4) The SSA repeats itself by collecting data in the new regions of the workspace until the setpoints can be shifted again (Fig. 6c). The procedure terminates by reaching the goal, leaving a (hyper-) ridge of data in space (Figure 6d).

The LQ controllers play a crucial role for devil sticking. Although we statistically exploit data thoroughly, it is nevertheless hard to build good local linear models in the high dimensional spaces, particularly at the beginning of learning. LQ control has useful robustness even if the underlying linear models are imprecise.

We tested the SSA in a noise corrupted simulation and on the real robot. Learning curves are given in Figure 7. The learning curves are typical for the given problem. It takes roughly 40 trials before the setpoint of each hand has moved close enough to the other hand's setpoint. For the simulation (Figure 7a) a break-through occurs and the robot rarely loses the devilstick after that. In Figure 7b, the real robot learning curve is shown. The real robot takes more trials to achieve longer juggling runs, and its performance is not very consistent. This was due to the fact that the stochasticities of the robot did not sample the full state space sufficiently well during the data collection phases of the SSA. As pointed out in the dual control paragraph of the SSA section, we now added some random noise to the controls generated by the LQ controllers. Figure 7c shows the remarkable improvement in performance. On average, human beings need roughly a week of 1 hour practicing a day before they learn to juggle the devilstick. With respect to this, the robot learned very quickly. But the stability of our controllers is not global so far and will require future work.

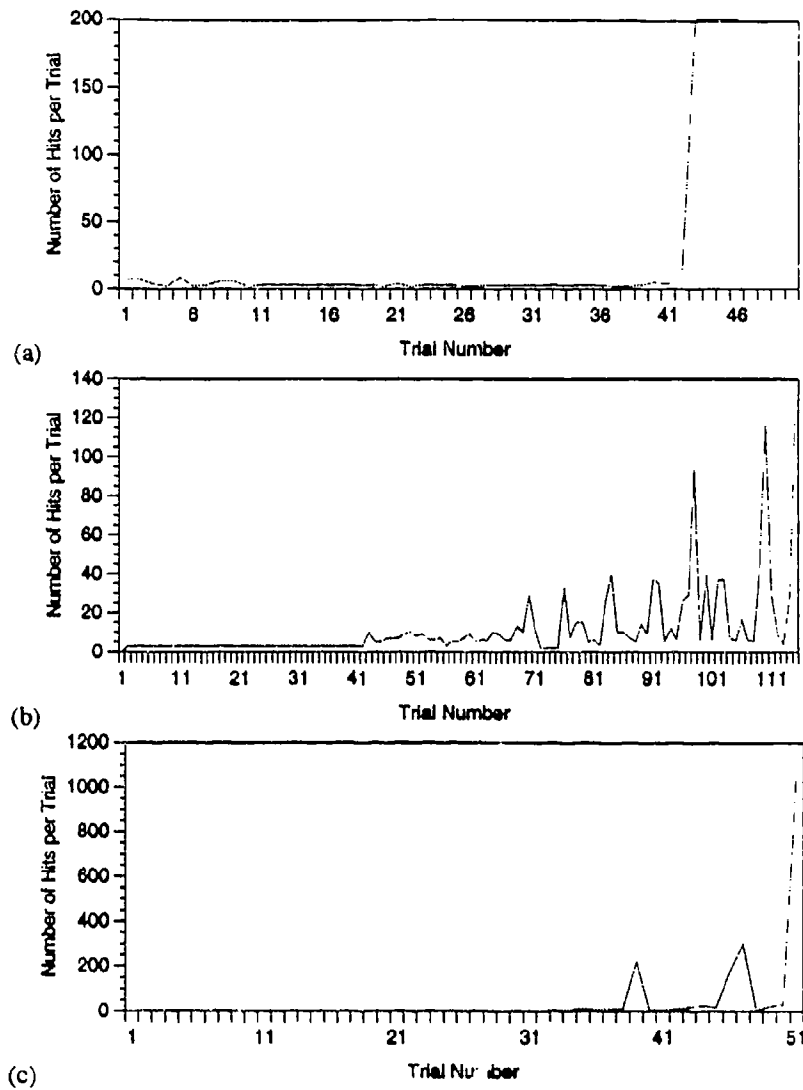


Figure 7: Learning curves of devil sticking using the SSA algorithm (a) simulation results (individual trials were stopped after 200 hits were reached), (b) real robot results; (c) real robot results with small amount of random noise added LQ controller commands

Discussion

In this paper we adopted a nonparametric approach to learning control. By means of locally weighted regression we built models of the world first, and exploited the models subsequently with statistical methods and algorithms from optimal control to design controllers. Despite the computational complexity of these methods, we demonstrated the usefulness of our algorithms in a real-time implementation of robot learning.

Using models for control according to the certainty equivalence principle is nothing new and has been supported by many researchers in the last years (e.g., [3, 38, 22, 24]). Using memory-based or nonparametric models, however, has only recently received increasing interest. One of the favorable advantages of memory-based modeling lies in the least commitment strategy which is associated with it. Since all data is kept in memory, a lookup can be optimized with respect to the few open architectural parameters. Parametric approaches do not have this ability if they discard their training data; if they retained all the training data they essentially become memory-based. As we demonstrated in our LWR approach to nonparametric modeling, several established statistical methods may be adopted to assess the quality of a model. These statistics form the backbone of the SSA exploration algorithm. So far we have only examined some of the most obvious statistical tools which directly relate to regression analysis. Many other methods may be suitable as well, particularly in a Bayesian framework.

Training a memory-based model is computationally inexpensive, as the data is simply stored in a memory. Training a nonlinear parametric model typically requires an iterative search for the appropriate parameters. Examples of iterative search are the various gradient descent techniques used to train neural network models (e.g., [20]). Lookup or evaluating a memory based model is computationally expensive, as described in this paper. Lookup for a nonlinear parametric model is often relatively inexpensive. If there is a situation in which a fixed set of training data is available, and there will be many queries to the model after the training data is processed, then it makes sense to use a nonlinear parametric model. However, if there is a continuous stream of new training data intermixed with queries, as there typically is in many motor learning problems, it may be less expensive to train and query a memory-based model than it is to train and query a nonlinear parametric model.

A question that often arises with memory-based models is the effect of memory limitations. We have not yet needed to address this issue in our experiments. However, we plan to explore how memory use can be minimized based on several methods. One approach is to only store "surprises". The system would try to predict the outputs of a data point before trying to store it. If the prediction is good, it is not necessary to store the point. Another approach is to forget data points. Points can be forgotten or removed from the database based on age, proximity to queries, or other criteria. Because memory-based learning retains the original training data, forgetting can be explicitly controlled.

That computational complexity does not necessarily limit real time applications was demonstrated with our successful devil sticking robot. We are able to do lookups for memory-based local models in less than 15ms for a thousand data points modeling a 10 to 5 mapping, and we are able to build on-line LQ controllers in another 5ms. The initial shortcomings of our deadbeat inverse or inverse-forward model controllers are not due to the

LWR learning algorithm but rather to the inherent problems of this kind of control. As has been pointed out by Jordan and Rumelhart [22], inverse models are not goal-directed and perform data sampling in action and not state space. They do not establish a connection between a certain sensation and a certain action but rather a connection between two sensations. Hence, they do not learn from bad actions. A forward model overcomes these problems. Pure forward model controllers, however, are still deadbeat controllers which try to cancel the plant dynamics in one step. This results in large commands if the system deviates only moderately from its desired goal and conflicts with the workspace bounds and command bounds of our robot. Additionally, modeling errors are strongly amplified by deadbeat control. Accurate data sampling, as it is necessary in high dimensional spaces, becomes thus rather difficult.

Due to the statistical properties of locally weighted regression, a simple exploration algorithm like the shifting setpoint algorithm is powerful enough to accomplish the desired task. Deadbeat control was replaced by LQ control which naturally blends into the LWR framework. By no means was the SSA algorithm intended to replace high-level controllers. Indeed, it remains to be explored in how far the chaining of individual LQ controllers is actually robust, and whether an approach from trajectory optimization [13] would not be more appropriate. In favor of the SSA algorithm stands its easy implementation for real-time systems.

Two crucial prerequisites entered our explanations on robot learning. First, we assumed we know the input/output representations of the task, and second, we were able to generate a goal state for the SSA exploration. A good choice of a representation is crucial in order to be able to accomplish the goal at all [33, 35], and we have very limited insight so far how to automate this part of the learning process. Of equal importance is a good choice of a goal state. In devil sticking, the goal state developed out of the necessity that the left and right hand have to cooperate. The initial action, however, which was given by the experimenter, clearly determined in which ballpark the juggling pattern would lie. Certain patterns of devil sticking are easier than others [34], and we picked an initial action of which we knew that it was favorable. One part of our future work will address these issues in more detail in that we search for good initial actions and strategies to approach a task [6].

Conclusions

The paper demonstrated that a real robot can indeed learn a non-trivial task. As pointed out above, by taking input/output representations and good learning goals as given, a large portion of the task was already solved in advance. Solving the remaining problems became practicable mainly because of the characteristics of the LWR learning method. The local linear models that this algorithm generated at every query point allowed us to make use of

optimal control techniques which added useful robustness to the controllers. Since LWR is memory-based, the local linear models could be optimized with respect to statistical uncertainty measures. These measures also served as the basis of the SSA exploration algorithm. Such statistical tools are not generally available in learning. LWR is particularly suited to exploit statistics since it originates from a statistical method, and we could thus easily assure compatibility of the statistics and the learning algorithm. As a last point, LWR does not suffer from problems of interference when being trained on new data. Interference means a degradation of performance in one part of the model when training the model with data relevant for different parts. Such an effect could happen during SSA shifting if a parametric learning method were applied. But since lookups with LWR are affected only by a small cloud of data in the neighborhood of the query point, interference problems are greatly reduced.

Our future work will focus on extending LWR model-based control to multistage problems in the optimal control domain. Devil sticking should not only be stable within the validity of the local linear controllers but rather exhibit global stability. This requires non-linear optimal control and planning techniques which we are currently exploring. Future work must furthermore address how we could approach tasks in which complete measurements of the states are not available, or what constitutes a state is not even known.

References

- [1] Aboaf, E. W., Atkeson, C. G., Reinkensmeyer, D. J. (1988), "Task-Level Robot Learning", *Proceedings of IEEE International Conference on Robotics and Automation*, April 24-29, Philadelphia, Pennsylvania (1988).
- [2] Aboaf, E.W., Drucker, S.M., Atkeson, C.G. (1989), "Task-Level Robot Learning: Juggling a Tennis Ball More Accurately", *Proceedings of IEEE International Conference on Robotics and Automation*, May 14-19, Scottsdale, Arizona (1989).
- [3] An, C.H., Atkeson, C.G., Hollerbach, J.M. (1988), *Model-Based Control of A Robot Manipulator*, Cambridge, MA: MIT Press (1988).
- [4] Åström, K.J., Wittenmark, B. (1989), *Adaptive Control*, Reading, MA: Addison-Wesley (1989).
- [5] Atkeson, C.G. (1990), "Memory-Based Approaches to Approximating Continuous Functions", MIT, Cambridge, MA: The AI-Lab and the Brain and Cognitive Sciences Department (1990).
- [6] Atkeson, C.G. (1994), "Using Local Trajectory Optimizers to Speed Up Global Optimization in Dynamic Programming", to appear in: Moody, J.E., Hanson, S.J., and Lippmann, R.P. (eds.) *Advances in Neural Information Processing Systems 6*, Morgan Kaufmann (1994).
- [7] Bar-Shalom, Y. (1981), "Stochastic Dynamic Programming: Caution and Probing", *IEEE Transactions on Automatic Control*, vol.AC-26, no.5, October 1981 (1981).
- [8] Bellman, R. (1957), *Dynamic Programming*, Princeton, N.J.: Princeton University Press (1957).
- [9] Bertsekas, D.P. (1987), *Dynamic Programming*, Englewood Cliffs, NJ: Prentice-Hall (1987).
- [10] Bitmead, R. R., Gevers, M., Wertz, V. (1990), *Adaptive Optimal Control*, Englewood Cliffs NJ: Prentice Hall (1990).

- [11] Cleveland, W.S., Devlin, S.J., Grosse, E. (1988), *Regression by Local Fitting: Methods, Properties, and Computational Algorithms*, Journal of Econometrics 37, 87-114, North-Holland (1988).
- [12] Duda, R.O., Hart, P.E. (1973), *Pattern Classification and Scene Analysis*, New York, NY: Wiley (1973).
- [13] Dyer, P., McReynolds, S.R. (1970), *The Computation and Theory of Optimal Control*, New York: Academic Press (1970).
- [14] Fan, J., Gijbels, I. (1992), "Variable Bandwidth And Local Linear Regression Smoothers", *The Annals of Statistics*, vol.20, no.4, pp.2008-2036 (1992).
- [15] Farmer, J.D., Sidorowich, J.J. (1987), "Predicting Chaotic Dynamics", Kelso, J.A.S., Mandell, A.J., Shlesinger, M.F., (eds.): *Dynamic Patterns in Complex Systems*, World Scientific Press (1987).
- [16] Farmer, J.D., Sidorowich, J.J. (1988), "Exploiting Chaos to Predict the Future and Reduce Noise", Technical Report LA-UR-88-901, Los Alamos National Laboratory, Los Alamos, NM (1988).
- [17] Fel'dbaum, A.A. (1965), *Optimal Control Systems*, New York, NY: Academic Press (1965).
- [18] Hampbell, F., Rousseeuw, P., Ronchetti, E., Stahel, W. (1985), *Robust Statistics*, Wiley International (1985).
- [19] Härdle, W. (1991), *Smoothing Techniques with Implementation in S*, New York, NY: Springer (1991).
- [20] Hertz, J., Krogh, A., Palmer, R.G. (1991), *Introduction to the Theory of Neural Computation*, Redwood City, CA: Addison Wesley (1991).
- [21] Isermann, R., Lachmann, K.-H., Matko, D. (1992), *Adaptive Control Systems*, New York, NY: Prentice Hall, (1992).
- [22] Jordan, I.M., Rumelhart, D.E. (1990), "Forward Models: Supervised Learning with a Distal Teacher", *MIT Center for Cognitive Science Occasional Paper 40*, Cambridge, MA: MIT (1990).
- [23] Mallows, C.L. (1966), "Choosing a Subset Regression", unpublished paper presented at the annual meeting of the American Statistical Association, Los Angeles (1966).
- [24] Moore, A. (1991), "Fast, Robust Adaptive Control by Learning only Forward Models", in: Moody, J.E., Hanson, S.J., and Lippmann, R.P. (eds.) *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann (1991).
- [25] Moore, A.W. (1990), *Efficient Memory-based Learning for Robot Control*, PhD. Thesis, Technical Report No. 229, Computer Laboratory, University of Cambridge, October 1990 (1990).
- [26] Moore, A.W. (1991), "Knowledge of Knowledge and Intelligent Experimentation for Learning Control", *Proceedings of the 1991 International Joint Conference on Neural Networks*, Seattle (1991).
- [27] Moore, A.W., Atkeson, C.G. (1993), "An Investigation of Memory-based Function Approximators for Learning Control", submitted to *Machine Learning* (1993).
- [28] Müller, H.-G. (1988), *Nonparametric Regression Analysis of Longitudinal Data*, Lecture Notes in Statistics Series, vol.46, Berlin: Springer (1988).
- [29] Myers, R.H. (1990), *Classical And Modern Regression With Applications*, Boston, MA: PWS-KENT (1990).
- [30] Narendra, K.S., Dorato, P. (eds.) (1991), *Advances in Adaptive Control*, Piscataway, NJ: IEEE Press (1991).
- [31] Press, W.P., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T. (1989), *Numerical Recipes in C - The Art of Scientific Computing*, Cambridge, MA: Press Syndicate University of Cambridge (1989).
- [32] Rizzi, A.A., Whitcomb, L.L., Koditschek, D.E. (1992a), "Distributed Real Time Control of a Spatial Robot Juggler", *IEEE Computer*, 25 (5) (1992).

- [33] Schaal, S., Atkeson, C.G. (1993b), "Open Loop Stable Control Strategies for Robot Juggling", *Proceedings IEEE International Conference on Robotics and Automation*, vol.3, pp.913-918, Georgia, Atlanta (1993).
- [34] Schaal, S., Atkeson, C.G., Botros, S. (1992b), "What Should Be Learned?", In: *Proceedings of Seventh Yale Workshop on Adaptive and Learning Systems*, pp.199-204, New Haven (1992).
- [35] Schaal, S., Sternad, D., (1993), "Learning Passive Motor Control Strategies with Genetic Algorithms", in: Nadel, L. & Stein, D. (eds.): *1992 Lectures in Complex Systems 1992*, Addison-Wesley (1993).
- [36] Slotine, J.-J.E., Li, W. (1991), *Applied Nonlinear Control*, Englewood Cliffs, NJ: Prentice Hall (1991).
- [37] Specht, D.F. (1991), "A General Regression Neural Network", *IEEE Transactions on Neural Networks*, vol.2, no.6, Nov.1991 (1991).
- [38] Sutton, R.S. (1990), "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming", *Proceedings of the International Machine Learning Conference*, pp.212-218 (1990).
- [39] Thrun, S.B. (1992a), "Efficient Exploration in Reinforcement Learning", Technical report CMU-CS-92-102, Pittsburgh, Pennsylvania: Carnegie-Mellon University (1992).
- [40] Van Zyl, G. (1991), "Design and control of a robotic platform for machine learning", MS thesis, MIT Dept. of Mechanical Engineering.
- [41] Wahba, G., Wold, S. (1975), "A Completely Automatic French Curve: Fitting Spline Functions By Cross-Validation", *Communications in Statistics*, 4(1) (1975)
- [42] White, D.A., Sofge, D.A. (1992), *Handbook of Intelligent Control : Neural, Fuzzy, and Adaptive Approaches*, New York, NY: Van Nostrand Reinhold (1992).

Appendix A

The conversion from unweighted linear regression to locally weighted regression analysis is done mostly by inserting the matrix W at the appropriate sites. The definitions of the parameters n' and p' , however, need some explanation. Imagine the weighting function is not a soft-weighting function (e.g., a Gaussian) but rather a hard-weighting function clipping off points beyond a certain threshold: $w_i = 1$ if $d^2 < \epsilon$, otherwise $w_i = 0$. Redefining n to n' accommodates such a k -nearest neighbor weighting and transforms the n -point regression problem to an k -point regression. If p stays unchanged, all statistics would correspond to unweighted regression. Subtraction of p from n to calculate variances aims at achieving unbiased estimators. For LWR, it is easily possible to find the case where $n' < p$. In the above mentioned k -nearest neighbor example, this would mean that we do not have sufficient data support for the regression model. For soft-weighting functions, the problem cannot be resolved so clearly; we could always imagine scaling up all weights by a constant factor so that $\sum_{i=1}^n w_i^2 = n$ which would not change the regression variables. LWR weights data with respect to each other and not absolutely. Applying n' instead of n to calculate variances or mean squared errors makes such measures invariant towards mere weight scaling; this can easily be verified by setting $w_i = w_i \cdot \text{const.}$. Defining p' in the given way avoids problems when $n' < p$. The bias introduced this way should diminish with an increasing number of data points in memory. One could argue in favor of neglecting p

entirely, but incorporating it in the given way makes the statistics stay on the more pessimistic side which seems reasonable.

The only statistics which see a direct influence of a weight scaling are the prediction intervals and the standardized individual PRESS residuals. Restricting the weights to the range $[0,1]$ and requiring that the weight of a point with zero Euclidean distance from the lookup point equals 1 resolves this problem. The dependence of the t-value on $n'-p'$ in the prediction intervals increases the t-value and thus the prediction intervals with diminishing n' . The smallest value t may acquire is for $n=n'$, i.e., unweighted regression. The standardized individual PRESS residual $e_{i,cross}$ is proportional to the magnitude of the i-th weight. As we pointed out in Section x3, this measure has zero mean and unit variance. With increasing distance from the current query point, it will be weighted down, i.e., the likelihood of being an outlier diminishes even if $e_{i,cross}$ is rather large. As the weights cannot be larger than 1, $e_{i,cross}$ cannot assume larger values as for unweighted regression.

It must be pointed out that statistics literature provides much more sophisticated and mathematically exact statistics for locally weighted regression [19, 28, 14, 11]. However, most of these measures require the estimation of Hessians and/or data densities which for high dimensional problems are not readily adapted without numerical problems. Our LWR statistics are used to tune fit parameters and need not give precise statistical assessments.

Table 1:

	Unweighted Linear Regression	Locally Weighted Regression
number of data in regression	n	n , we define: $n' = \sum_{i=1}^n w_i^2$, $w_i \in [0,1]$, $w_i = 1$ if $\mathbf{x}_i = \mathbf{x}_q$, $\mathbf{x}_q$ is the current query point
number of regression parameters	p	p , we define: $p' = \frac{n'}{n} p$
matrix and vector definitions	$\mathbf{x}_i = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{p-1} \\ 1 \end{bmatrix}$, $\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix}$	$\mathbf{x}_i = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{p-1} \\ 1 \end{bmatrix}$, $\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix}$, $\mathbf{W} = \begin{bmatrix} w_1 & & & 0 \\ & w_2 & & \\ & & \ddots & \\ 0 & & & w_n \end{bmatrix}$
regression model	$\mathbf{X}\beta = \mathbf{y}$	$\mathbf{W}\mathbf{X}\beta = \mathbf{W}\mathbf{y}$

sum squared error = residual error	$SSE_{res} = (\mathbf{X}\beta - \mathbf{y})^T (\mathbf{X}\beta - \mathbf{y})$	$SSE_{res} = (\mathbf{X}\beta - \mathbf{y})^T \mathbf{W}^T \mathbf{W} (\mathbf{X}\beta - \mathbf{y})$
normal equations	$(\mathbf{X}^T \mathbf{X})\beta = \mathbf{X}^T \mathbf{y}$	$(\mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{X})\beta = \mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{y}$
solution for regression parameters	$\beta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$	$\beta = (\mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{y}$
variance of residuals	$s^2 = \frac{(\mathbf{X}\beta - \mathbf{y})^T (\mathbf{X}\beta - \mathbf{y})}{n - p}$	$s^2 = \frac{(\mathbf{X}\beta - \mathbf{y})^T \mathbf{W}^T \mathbf{W} (\mathbf{X}\beta - \mathbf{y})}{n' - p'}$
mean squared cross validation error (mean PRESS residual)	$MSE_{cross} = \frac{1}{n - p} \sum_{i=1}^n \left(\frac{y_i - \mathbf{x}_i^T \beta}{1 - \mathbf{x}_i^T (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{x}_i} \right)^2$	$MSE_{cross} = \frac{1}{n' - p'} \sum_{i=1}^n \left(\frac{w_i (y_i - \mathbf{x}_i^T \beta)}{1 - w_i \mathbf{x}_i^T (\mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{x}_i w_i} \right)^2$
prediction intervals	$I_i = \mathbf{x}_i^T \beta \pm t_{\alpha/2, n-p} s \sqrt{1 + \mathbf{x}_i^T (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{x}_i}$ where $t_{\alpha/2, n-p}$ is Student's t-value of n-p degrees of freedom for a 100(1-α)% prediction bound	$I_i = \mathbf{x}_i^T \beta \pm t_{\alpha/2, n'-p'} s \sqrt{1 + \mathbf{x}_i^T (\mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{x}_i}$ where $t_{\alpha/2, n'-p'}$ is Student's t-value of n'-p' degrees of freedom for a 100(1-α)% prediction bound
standardized individual PRESS residual	$e_{i, cross} = \frac{y_i - \mathbf{x}_i^T \beta}{s \sqrt{1 - \mathbf{x}_i^T (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{x}_i}}$	$e_{i, cross} = \frac{w_i (y_i - \mathbf{x}_i^T \beta)}{s \sqrt{1 - w_i \mathbf{x}_i^T (\mathbf{X}^T \mathbf{W}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{x}_i w_i}}$

Memory-Based Neural Networks For Robot Learning

Christopher G. Atkeson and Stefan Schaal

College of Computing

Georgia Institute of Technology

801 Atlantic Drive, Atlanta, GA 30332-0280

cga@cc.gatech.edu, sschaal@ai.mit.edu

Abstract

This paper explores a memory-based approach to robot learning, using memory-based neural networks to learn models of the task to be performed. Steinbuch and Taylor presented neural network designs to explicitly store training data and do nearest neighbor lookup in the early 1960s. In this paper their nearest neighbor network is augmented with a local model network, which fits a local model to a set of nearest neighbors. This network design is equivalent to a statistical approach known as locally weighted regression, in which a local model is formed to answer each query, using a weighted regression in which nearby points (similar experiences) are weighted more than distant points (less relevant experiences).

The memory-based neural network architecture can represent smooth nonlinear functions, yet has simple training rules with a single global optimum. The paper explains how an appropriate distance metric or measure of similarity can be found, and how the distance metric is used. We localize the architectural parameters of the approach, so that parameters such as distance metrics are a function of the current query point instead of being global. The paper also explains how irrelevant input variables and terms in the local model are detected. Statistical tests are presented for when a local model is good enough and sampling should be moved to a new area. Our methods explicitly deal with the case where prediction accuracy requirements exist during exploration. By gradually shifting a center of exploration and controlling the speed of the shift based on local prediction accuracy, a goal-directed exploration of state space takes place along the fringes of the current data support until the task goal is achieved.

We illustrate this approach by describing how it has been used to enable a robot to learn a difficult juggling task.

1 Introduction

An important problem in motor learning is approximating a continuous function from samples of the function's inputs and outputs. This paper explores a neural network architecture that simply remembers experiences (samples) and builds a local model to answer any particular query (an input for which the function's output is desired). Steinbuch (Steinbuch 1961, Steinbuch and Piske 1963) and Taylor (Taylor 1959, Taylor 1960) independently proposed neural network designs that explicitly

remembered the training experiences and used a local representation to do nearest neighbor lookup. They pointed out that this approach could be used for control. They used a layer of hidden units to compute an inner product of each stored vector with the input vector. A winner-take-all circuit then selected the hidden unit with the highest activation. This type of network can find nearest neighbors or best matches using a Euclidean distance metric (Kazmierczak and Steinbuch 1963). In this paper their nearest neighbor lookup network (which I will refer to as the memory network) is augmented with a local model network, which fits a local model to a set of nearest neighbors.

The memory-based neural network design can represent smooth nonlinear functions, yet has simple training rules with a single global optimum for building a local model in response to a query. Our philosophy is to model complex continuous functions using simple local models. This approach avoids the difficult problem of finding an appropriate structure for a global model and allows complex nonlinear models to be identified (trained) quickly. A key idea is to form a training set for the local model network after a query to be answered is known. This approach allows us to include in the training set only relevant experiences (nearby samples), and to weight the experiences according to their relevance to the query. The local model network, which may be a simple network architecture such as a perceptron, forms a model of the portion of the function near the query point, much as a Taylor series models a function in a neighborhood of a point. This local model is then used to predict the output of the function, given the input. After answering the query, a new local model is trained to answer the next query. This approach minimizes interference between old and new data, and allows the range of generalization to depend on the density of the samples.

Currently we are using polynomials as the local models. Since the polynomial local models are linear in the unknown parameters, we can estimate these parameters using a linear regression. We use cross validation to choose an appropriate distance metric and weighting function, and to help find irrelevant input variables and terms in the local model. In this approach cross validation is no more computationally expensive than answering a query. This is quite different from a parametric neural network, where a new network must be trained for each cross validation training set. We extend this approach to give information about the reliability of the predictions and local linearizations generated by locally weighted regression. This allows the robot to monitor its own skill level and guide its exploratory behavior. The polynomial local models allow us to efficiently estimate local linear models for different points in the state space. These local linear models are used in several ways during learning.

We use several forms of indirect learning, where a model is learned and then control actions are chosen based on the model, rather than direct learning, where appropriate control actions are learned directly. Our starting point for modeling is that we do not know the structure or form of the system to be controlled. We assume we do know what constitutes a state of the system, and that we measure the complete state. Later papers will discuss how we could approach tasks in which complete measurements of the state are not available, or what constitutes a state is not even known.

The learned models are multidimensional functions that are approximated from sampled data (the previous experiences or attempts to perform the task). Goals for function approximation in robot learning go beyond being able to represent the training set and generalize appropriately. The learned models are used in a variety of ways to successfully execute the task. We would like the models to incorporate the latest information. The models will be continuously updated with a stream of new training data, so updating a model with new data should take a short period of time. There are also time constraints on how long it can take to use a model to make a prediction. Because we are interested in control methods that make use of local linearizations of the plant model, we want a

representation that can quickly compute a local linear model of the represented transformation. We also need to be able to find first (and potentially second) derivatives of the learned function. We would like to minimize the negative interference from learning new knowledge on previously stored information. The ability to tell where in the input space the function is accurately approximated is very useful. This is typically based on the local density of samples, and an estimate of the local variance of the outputs. This ability is used in iterative use of the model to determine when to terminate search and collect more data.

2 Locally Weighted Regression

As the most generic approximator that satisfies many of these criteria, we are exploring a version of memory-based learning technique called locally weighted regression (LWR) (Cleveland et al. 1988, Atkeson 1990). A memory-based learning (MBL) system is trained by storing the training data in a memory. This allows MBL systems to achieve real-time learning. MBL avoids interference between new and old data by retaining and using all the data to answer each query. MBL approximates complex functions using simple local models, as does a Taylor series. Examples of types of local models include nearest neighbor, weighted average, and locally weighted regression (Figure 1). Each of these local models combine points near to a query point to estimate the appropriate output. Nearest neighbor local models simply choose the closest point and use its output value. Weighted average local models sum the outputs of nearby points weighted by their distance to the query point. Locally weighted regression fit a surface to nearby points using a distance weighted regression.

The weights in the locally weighted regression depend on parameters used to calculate a distance metric and a weighting function, and stabilize the solution to the regression. We will refer to these parameters as "architectural parameters". These parameters can be optimized automatically in a local fashion using cross validation.

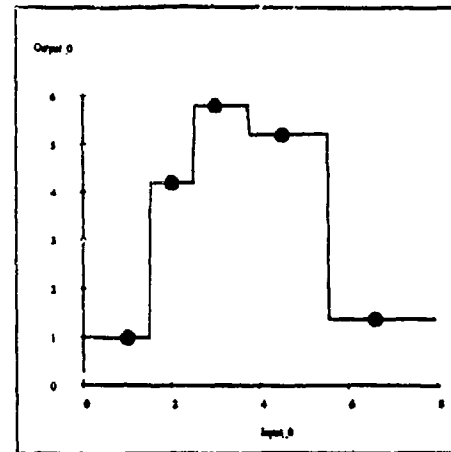
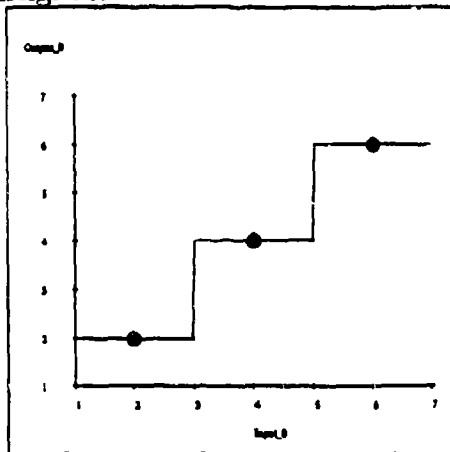
Locally weighted regression uses a relatively complex regression procedure to form the local model, and is thus more expensive than nearest neighbor and weighted average memory-based learning procedures. For each query a new local model is formed. The rate at which local models can be formed and evaluated limits the rate at which queries can be answered. This section describes how locally weighted regression can be implemented in real time.

2.1 An example

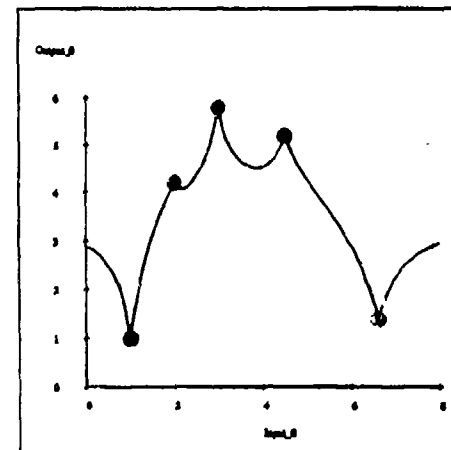
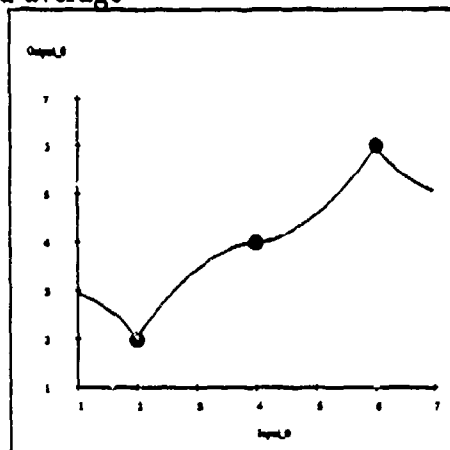
As an example of modeling a function using locally weighted regression, we will consider a problem from motor control and robotics, two-joint arm inverse dynamics. We will predict torques at the shoulder, τ_1 , and elbow, τ_2 , on the basis of joint positions, θ_1 and θ_2 , joint velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$, and joint accelerations, $\ddot{\theta}_1$ and $\ddot{\theta}_2$. We use this example because we already know the idealized function, and will be able to assess how well the locally weighted regression procedure is doing and interpret the parameters used to improve the fit. In an actual application a structured model (An, Atkeson, and Hollerbach 1988, for example) would be used to fit the dynamics data, and the locally weighted regression would be used to fit the errors (residuals) of the structured model.

We have a query point $(\theta_1^*, \theta_2^*, \dot{\theta}_1^*, \dot{\theta}_2^*, \ddot{\theta}_1^*, \ddot{\theta}_2^*)$ for which we want to predict the shoulder and elbow torques. We will first show how an unweighted regression can be used to form a global model. Then we will show how a weighted regression can be used to form a local model appropriate to answer this

Nearest neighbor



Weighted average



Locally weighted regression

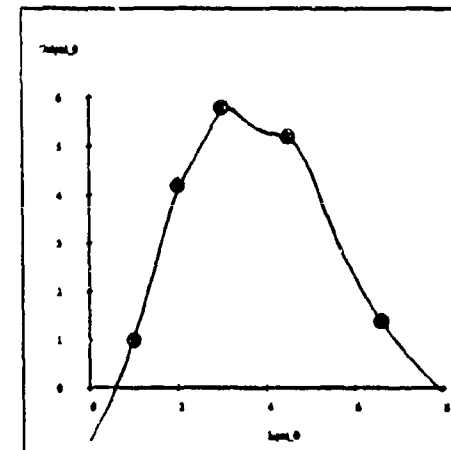
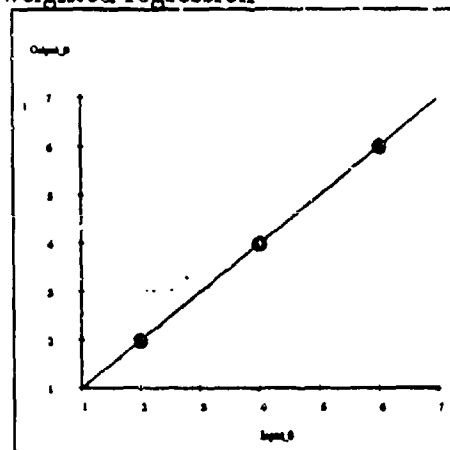


Figure 1: Fits using different types of local models for three and five data points.

particular query. For the purposes of this example we will assume a quadratic model is used in the regression. In this dynamics example there are 28 terms in the quadratic model:

$$\begin{array}{ccccccc}
 1 & \theta_1, & \theta_2, & \dot{\theta}_1, & \dot{\theta}_2, & \ddot{\theta}_1, & \ddot{\theta}_2, \\
 & \theta_1 * \theta_1, & \theta_1 * \theta_2, & \theta_1 * \dot{\theta}_1, & \theta_1 * \dot{\theta}_2, & \theta_1 * \ddot{\theta}_1, & \theta_1 * \ddot{\theta}_2, \\
 & & \theta_2 * \theta_2, & \theta_2 * \dot{\theta}_1, & \theta_2 * \dot{\theta}_2, & \theta_2 * \ddot{\theta}_1, & \theta_2 * \ddot{\theta}_2, \\
 & & & \dot{\theta}_1 * \dot{\theta}_1, & \dot{\theta}_1 * \dot{\theta}_2, & \dot{\theta}_1 * \ddot{\theta}_1, & \dot{\theta}_1 * \ddot{\theta}_2, \\
 & & & & \dot{\theta}_2 * \dot{\theta}_2, & \dot{\theta}_2 * \ddot{\theta}_1, & \dot{\theta}_2 * \ddot{\theta}_2, \\
 & & & & & \ddot{\theta}_1 * \ddot{\theta}_1, & \ddot{\theta}_1 * \ddot{\theta}_2, \\
 & & & & & & \ddot{\theta}_2 * \ddot{\theta}_2
 \end{array}$$

where 1 represents the constant term in the model.

Let us assume we have 1000 samples of the two joint arm dynamics function. To form a local model of the shoulder torque involves finding estimates of the 28 terms or parameters of the local quadratic model. The equation to be solved is

$$\mathbf{X}\beta = \mathbf{y} \quad (1)$$

where $\mathbf{X}$ is a 1000 by 28 data matrix, in which each row has the 28 terms of the quadratic model corresponding to a point (sample of the function), and each column corresponds to a particular term in the quadratic model. β is the vector of 28 estimated parameters of the quadratic model, and $\mathbf{y}$ is the vector of 1000 shoulder torques from the 1000 points included in the regression.

An unweighted regression finds the solution to the normal equations:

$$(\mathbf{X}^T \mathbf{X}) \beta = \mathbf{X}^T \mathbf{y} \quad (2)$$

The estimated parameters are used, with the query point, to predict the shoulder torque for the query point. Another set of parameters are estimated for the elbow torque.

However, we assume the global quadratic model is not the correct model structure for predicting the torques. These structural modeling errors imply that different sets of parameters are estimated by the regression, given different data sets. The data set can be tailored to the query point by emphasizing nearby points in the regression. The origin of the input data is first shifted by subtracting the query point from each data point. Then each data point is given a weight.

Unweighted regression gives distant points equal influence with nearby points on the ultimate answer to the query, for equally spaced data. To weight similar points more, locally weighted regression is used. First, a distance is calculated from each of the stored data points (rows in the $\mathbf{X}$ matrix) to the query point $\mathbf{q}$:

$$d_i^2 = \sum_j m_j (\mathbf{X}_{ij} - \mathbf{q}_j)^2 \quad (3)$$

For the robot arm dynamics example, d_i^2 is calculated for each point in the following way:

$$\begin{aligned}
 d^2 = & m_1^2(\theta_1 - \theta_1^*)^2 + m_2^2(\theta_2 - \theta_2^*)^2 + m_3^2(\dot{\theta}_1 - \dot{\theta}_1^*)^2 \\
 & + m_4^2(\dot{\theta}_2 - \dot{\theta}_2^*)^2 + m_5^2(\ddot{\theta}_1 - \ddot{\theta}_1^*)^2 + m_6^2(\ddot{\theta}_2 - \ddot{\theta}_2^*)^2
 \end{aligned} \quad (4)$$

The superscript * indicates the query point, and the m_j are the components of the distance metric. The weight for each stored data point is a function of that distance:

$$w_i = f(d_i^2) \quad (5)$$

Each row i of $\mathbf{X}$ and $\mathbf{y}$ is multiplied by the corresponding weight w_i . A simple weighting function just raises the distance to a negative power. The magnitude of the power determines how local the regression will be (the rate of dropoff of the weights with distance).

$$w_i = \frac{1}{d_i^p} \quad (6)$$

This type of weighting function goes to infinity as the query point approaches a stored data point. This forces the locally weighted regression to exactly match that stored point. If the data is noisy, exact interpolation is not desirable, and a weighting scheme with limited magnitude is desired. One such scheme, which we use in implementations on actual robots, is a Gaussian kernel:

$$w_i = \exp\left(\frac{-d_i^2}{2k^2}\right) \quad (7)$$

The parameter k scales the size of the kernel to determine how local the regression will be.

A potential problem is that the data points may be distributed in such a way as to make the regression matrix $\mathbf{X}$ nearly singular. Ridge regression (Draper and Smith 1981) is used to prevent problems due to a singular data matrix. The following equation, with $\mathbf{X}$ and $\mathbf{y}$ already weighted, is solved for β :

$$(\mathbf{X}^T \mathbf{X} + \Lambda) \beta = \mathbf{X}^T \mathbf{y} \quad (8)$$

where Λ is a diagonal matrix with small positive diagonal elements λ_i^2 . This is equivalent to adding i extra rows to $\mathbf{X}$, each having a single non-zero element, λ_i , in the i th column. Adding additional rows can be viewed as adding "fake" data, which, in the absence of sufficient real data, biases the parameter estimates to zero (Draper and Smith 1981). Another view of ridge regression parameters is that they are the Bayesian assumptions about the apriori distributions of the estimated parameters (Seber 1977).

2.2 Assessing the computational cost

Lookup has three stages: forming weights, forming the regression matrix, and solving the normal equations. Let us examine how the cost of each of these stages grows with the size of the data set and dimensionality of the problem. We will assume a linear local model.

Forming and applying the weights involves scanning the entire data set, so it scales linearly with the number of data points in the database (n). For each of d input dimensions there are a constant number of operations, so the number of operations scales linearly with the number of input dimensions. Note that we can eliminate points whose distance is above a threshold, reducing the number of points considered in subsequent stages of the computation.

Each element of $\mathbf{X}^T \mathbf{X}$ and $\mathbf{X}^T \mathbf{y}$ is the inner (dot) product of two columns of $\mathbf{X}$ or $\mathbf{y}$. The architecture of digital signal processors is ideally suited for this computation, which consists of repeated multiplies and accumulates. The computation is linear in the number of rows n and quadratic in the number of columns ($d^2 + d * o$), where d is the number of input dimensions and o is the number of output dimensions.

Solving the normal equations is done using a LDL^T decomposition, which is cubic in the number of input dimensions, and independent of the number of data points. Other more sophisticated and

more expensive decompositions, such as the singular value decomposition, do not need to be used since the ridge regression procedure guarantees that the normal equations will be well-conditioned.

The most straightforward parallel implementation of locally weighted regression would distribute the data points among several processors. Queries can be broadcast to the processors, and each processor can weight its data set and form its contribution to $X^T X$ and $X^T y$. These contributions can be summed and the full normal equations solved on a single processor. The communication costs are linear in the number of processors and quadratic in the number of columns ($d^2 + d * o$), and independent of the total number of points.

We have implemented the local weighted regression procedure on a 33MHz Intel i860 micro-processor. The peak computation rate of this processor is 66 MFlops. We have achieved effective computation rates of 20 MFlops on a learning problem with 10 input dimensions and 5 output dimensions, using a linear local model. This leads to a lookup time of approximately 20 milliseconds on a database of 1000 points.

This memory-based approach can also be simulated using k-d tree data structures (Friedman, Bentley, and Finkel 1977) on a standard serial computer and using parallel search on a massively parallel computer, the Connection Machine (Hillis 1985).

2.3 Implementing locally weighted regression in a neural network

The memory network of Steinbuch and Taylor can be used to find the nearest stored vectors to the current input vector. The memory network computes a measure of the distance between each stored vector and the input vector in parallel, and then a "winner take all" network selects the nearest vector (nearest neighbor). Euclidean distance has been chosen as the distance metric, because the Euclidean distance is invariant under rotation of the coordinates used to represent the input vector.

The memory network consists of three layers of units: input units, hidden or memory units, and output units. The input units are fully connected to the hidden units. The squared Euclidean distance between the input vector (i) and a weight vector (w_k) for the connections of the input units to hidden unit k is given by:

$$d_k^2 = (i - w_k)^T (i - w_k) = i^T i - 2i^T w_k + w_k^T w_k$$

Since the quantity $i^T i$ is the same for all the hidden units, minimizing the distance between the input vector and the weight vector for each hidden unit is equivalent to maximizing:

$$i^T w_k - 1/2 w_k^T w_k$$

This quantity is the inner product of the input vector and the weight vector for hidden unit k , biased by half the squared length of the weight vector. Maximizing this quantity using a winner take all circuit allows the unit with the smallest distance to be selected.

Dynamics of the memory network neurons allow the memory network to output a sequence of nearest neighbors. These nearest neighbors form the selected training sequence for the local model network. Memory unit dynamics can also be used to allocate "free" memory units to new experiences, and to forget old training points when the capacity of the memory network is fully utilized.

The local model network consists of only one layer of modifiable weights preceded by any number of layers with fixed connections. There may be arbitrary preprocessing of the inputs of the local model, but the local model is linear in the parameters used to form the fit. The local model network

using the LMS training algorithm performs a linear regression of the transformed inputs against the desired outputs. Thus, the local model network can be used to fit a linear regression model to the selected training set. With multiplicative interactions between inputs the local model network can be used to fit a polynomial surface (such as a quadratic) to the selected training set. An alternative implementation of the local model network could use a single layer of "sigma-pi" units (Durbin and Rumelhart 1989).

This network design has simple training rules. In the memory network the weights are simply the values of the components of input and output vectors, and the bias for each memory unit is just half the squared length of the corresponding input weight vector. No search for weights is necessary, since the weights are directly given by the data to be stored. The local model network is linear in the weights, leading to a single optimum which can be found by linear regression or gradient descent. Thus, convergence to the global optimum is guaranteed when forming a local model to answer a particular query.

3 Related Work

Memory-based representations have a long history. Approaches which represent previous experiences directly and use a similar experience or similar experiences to form a local model are often referred to as nearest neighbor or k-nearest neighbor approaches. Local models (often polynomials) have been used for many years to smooth time series (Sheppard 1912, Sherriff 1920, Whittaker and Robinson 1924, Macauley 1931) and interpolate and extrapolate from limited data. Barnhill (1977) and Sabin (1980) survey the use of nearest neighbor interpolators to fit surfaces to arbitrarily spaced points. Eubank (1988) surveys the use of nearest neighbor estimators in nonparametric regression. Lancaster and Salkauskas (1986) refer to nearest neighbor approaches as "moving least squares" and survey their use in fitting surfaces to data. Farmer and Sidorowich (1988a, 1988b) survey the use of nearest neighbor and local model approaches in modeling chaotic dynamic systems. Kawamura and Nakagawa (1990) and Kawamura, Noborio, and Nakagawa (1990) describe approaches to memory-based control of robots. Specht (1991) describes a memory-based neural network approach based on a probabilistic model that motivates using weighted averaging as the local model.

An early use of direct storage of experience was in pattern recognition. Fix and Hodges (1951, 1952) suggested that a new pattern could be classified by searching for similar patterns among a set of stored patterns, and using the categories of the similar patterns to classify the new pattern. Steinbuch and Taylor proposed a neural network implementation of the direct storage of experience and nearest-neighbor search process for pattern recognition (Steinbuch 1961, Taylor 1959), and pointed out that this approach could be used for control (Steinbuch and Piske 1963). Stanfill and Waltz (1986) proposed using directly stored experience to learn pronunciation, using a Connection Machine and parallel search to find relevant experience. They have also applied their approach to medical diagnosis (Waltz 1987) and protein structure prediction.

Nearest neighbor approaches have also been used in nonparametric regression and fitting surfaces to data. Often, a group of similar experiences, or nearest neighbors, is used to form a local model, and then that model is used to predict the desired value for a new point. Local models are formed for each new access to the memory. Watson (1964), Royall (1966), Crain and Bhattacharyya (1967), Cover (1968), and Shepard (1968) proposed using a weighted average of a set of nearest neighbors. Gordon and Wixom (1978) and Barnhill, Dube, and Little (1983) analyze such weighted average

schemes. Crain and Bhattacharyya (1967), Falconer (1971), and McLain (1974) suggested using a weighted regression to fit a local polynomial model at each point a function evaluation was desired. All of the available data points were used. Each data point was weighted by a function of its distance to the desired point in the regression. McIntyre, Pollard, and Smith (1968), Peltó, Elkins, and Boyd (1968), Legg and Brent (1969), Palmer (1969), Walters (1969), Lodwick and Whittle (1970), Stone (1975), and Franke and Nielson (1980) suggested fitting a polynomial surface to a set of nearest neighbors, also using distance weighted regression. Stone scaled the values in each dimension when the experiences were stored. The standard deviations of each dimension of previous experiences were used as the scaling factors, so that the range of values in each dimension were approximately equal. This affects the distance metric used to measure closeness of points. Cleveland (1979) proposed using robust regression procedures to eliminate outlying or erroneous points in the regression process. A program implementing a refined version of this approach (LOESS) is available by sending electronic mail containing the single line, *send dloess from a*, to the address netlib@research.att.com (Grosse 1989). Cleveland, Devlin and Grosse (1988) analyze the statistical properties of the LOESS algorithm and Cleveland and Devlin (1988) show examples of its use. Stone (1977, 1982), Devroye (1981), Lancaster (1979), Lancaster and Šalkauskas (1981), Cheng (1984), Li (1984), Farwig (1987), and Müller (1987) provide analyses of nearest neighbor approaches. Franke (1982) compares the performance of nearest neighbor approaches with other methods for fitting surfaces to data.

4 Learning in simulation

The network has been used for motor learning of a simulated arm and a simulated running machine. The network performed surprisingly well in these simple evaluations. The simulated arm was able to follow a desired trajectory after only a few practice movements. Performance of the simulated running machine in following a series of desired velocities was also improved. This paper will report only on the arm trajectory learning.

Figure 2 shows the simulated 2-joint planar arm. The problem faced in this simulation is to learn the correct joint torques to drive the arm along the desired trajectory (the inverse dynamics problem). In addition to the feedforward control signal produced by the network described in this paper, a feedback controller was also used.

Figure 3 shows several learning curves for this problem. The first point in each of the curves shows the performance generated by the feedback controller alone. The error measure is the RMS torque error during the movement. The highest curve shows the performance of a nearest neighbor method without a local model. On each time step the nearest point was used to generate the torques for the feedforward command, which were then summed with the output from the feedback controller. The second curve shows the performance using a linear local model. The third curve shows the performance using a quadratic local model. Adding the local model network greatly speeds up learning. The network with the quadratic local model learned more quickly than the one with the local linear model.

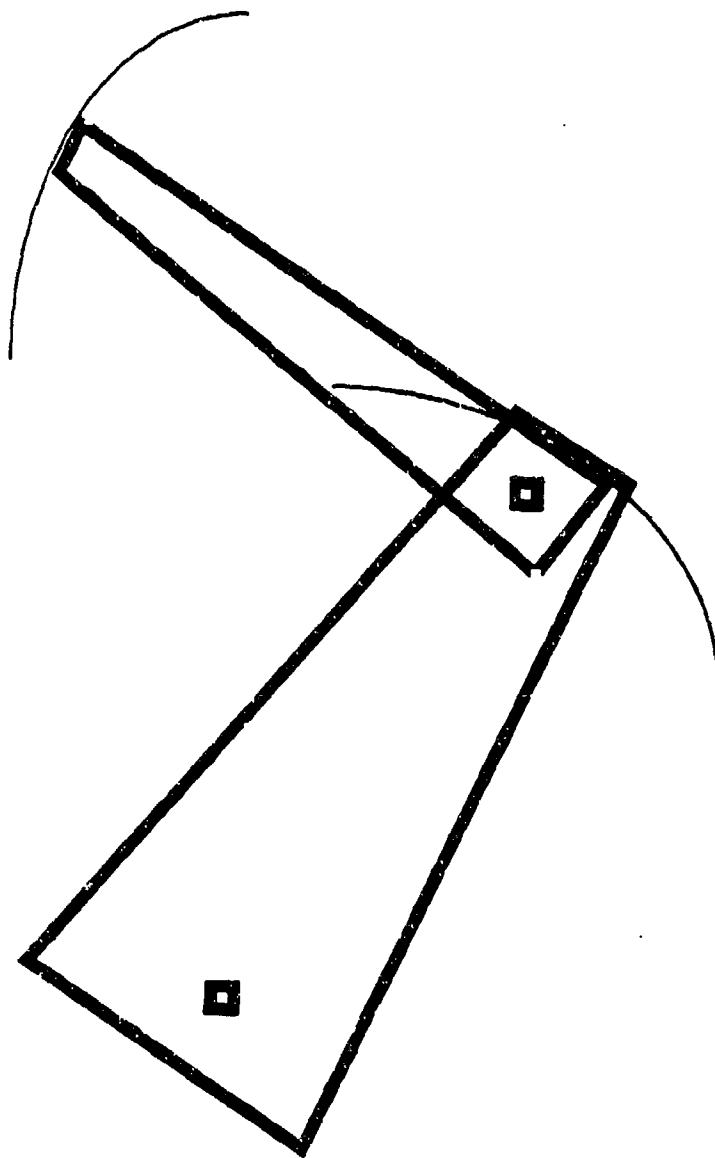
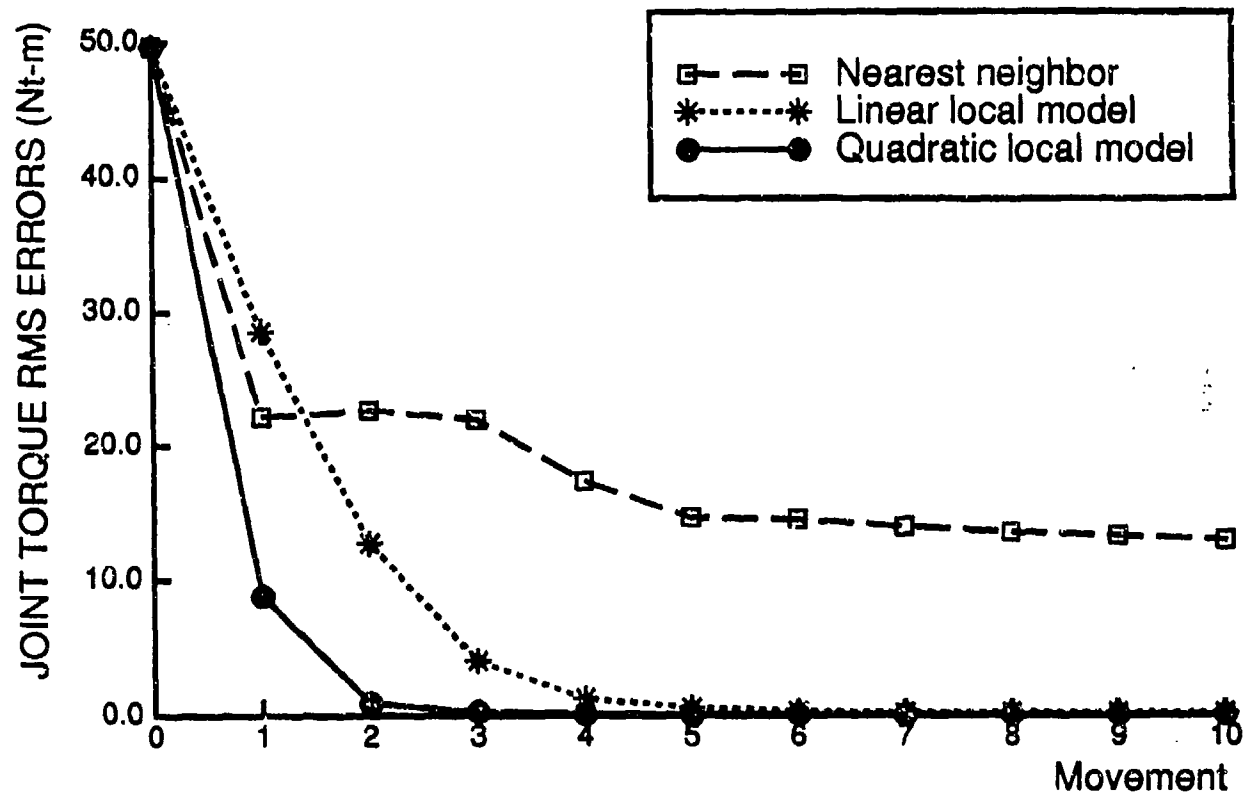


Figure 2: Simulated Planar Two-joint Arm



2 Joint Arm Trajectory Learning.

Figure 3: Learning curves from 3 different network designs on the two joint arm trajectory learning problem.

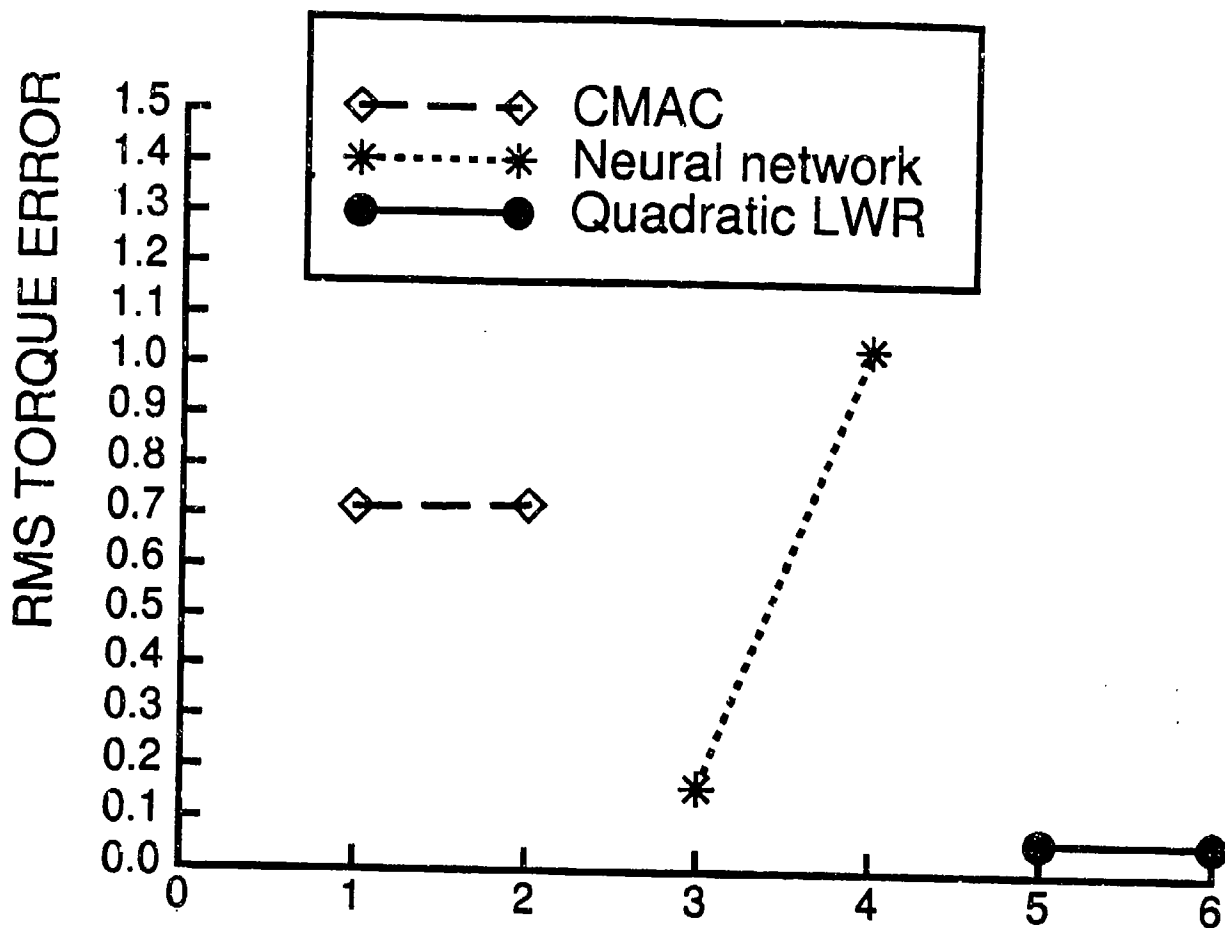


Figure 4: Performance of various methods on two joint arm dynamics.

5 Interference

To illustrate the differences between some proposed neural network representations and a memory-based representation, two neural network methods, CMAC (Albus 1975ab) and sigmoidal feedforward neural networks, were compared to the approach explored in this paper. The parameters for the CMAC approach were taken from Miller, Glanz, and Kraft (1987) who used the CMAC to model arm inverse dynamics. The architecture for the sigmoidal feedforward neural network was taken from Goldberg and Pearlmuter (1988, section 6) who also modeled arm inverse dynamics.

The ability of each of these methods to predict the torques of the simulated two joint arm at 1000 random points was compared. Figure 4 plots the normalized RMS prediction error. The points were sampled uniformly using ranges comparable to those used in (Miller et al 1987). Initially, each method was trained on a training set of 1000 random samples of the two joint arm dynamics function, and then the predictions of the torques on a separate test set of 1000 random samples of the two joint arm dynamics function were assessed (points 1, 3, and 5). Each method was then trained on 10 attempts to make a particular desired movement. Each method successfully learned

the desired movement. After this second round of training, performance on the random test set was again measured (points 2, 4, and 6).

The data indicate that the locally weighted regression approach (filled in circles) and the sigmoidal feedforward network approach (asterisks) both generalize well on this problem (points 3 and 5 have low error). The CMAC (diamonds) did not generalize well on this problem (point 1 has a large error), although it represented the original training set with a normalized RMS error of 0.000001. A variety of CMAC resolutions were explored, ranging from a basic CMAC cell size covering the entire range of data to a cell size covering a fifth of the data range in each dimension. A cell size covering one half the data ranges in each dimension generalized best (the data shown here).

After training on a different training set (the attempts to make a particular desired movement), the sigmoidal feedforward neural network lost its memory of the full dynamics (point 4), and represented only the dynamics of the particular movements being learned in the second training set. This interference between new and previously learned data was not prevented by increasing the number of hidden units in the single layer network from 10 up to 100. The other methods explored did not show this interference effect (points 2 and 6).

6 Tuning Architectural Parameters Globally

For the example problem of two joint arm inverse dynamics, we have introduced 34 free parameters into the local regression process: the weighting function dropoff parameter p , the 6 elements of the distance metric m_i , and the 27 variable diagonal elements of Λ (the ridge regression parameters λ_i). The element of Λ corresponding to the constant term, λ_1 , is held fixed.

A cross validation approach is used to choose values for these fit parameters. For each point a query is done to estimate the output at that point, after removing the point from the database. The difference between the estimate and the actual value for that point is the cross validation error for that point. The sum of the squared cross validation errors is minimized using a nonlinear parameter estimation procedure (MINPACK (More, Garbow, and Hillstom 1980) or NL2SOL (Dennis, Gay, and Welsch 1981), for example). Because the local model is linear in the unknown parameters we can analytically take the derivative of the cross validation error with respect to the parameters to be estimated, which greatly speeds up the search process. In the memory-based approach computing the cross validation error for a single point is no more computationally expensive than answering a query. This is quite different from a parametric neural network, where a new network must be trained for each cross validation training set with a particular point removed.

The cross validation to optimize the fit parameters may be done globally, using all the experiences in the memory to produce one set of fit parameters. Different fit parameters can be used for different outputs. The cross validation may also be done locally, either with each query, or separately for different regions of the input space, producing different sets of fit parameters specialized for particular queries, as discussed in the next section.

We can use the optimized distance metric to find which input variables are irrelevant to the function being represented. In the horizontal two-joint arm inverse dynamics problem, m_1 , the weight on the distances in the θ_1 direction typically drops to zero, indicating that the input variable θ_1 is irrelevant to predicting τ_1 and τ_2 . This is actually the case, as θ_1 does not appear in the true dynamics equations for an arm operating in a horizontal plane.

We can also interpret the ridge regression parameters, λ_i . Since the arm dynamics are linear in

acceleration, the terms in the local model that are quadratic in acceleration ($\ddot{\theta}_1^2, \ddot{\theta}_1 * \ddot{\theta}_2, \ddot{\theta}_2^2$) are not relevant to predicting torques. Similarly the products of velocity and acceleration ($\dot{\theta}_1 * \ddot{\theta}_1, \dot{\theta}_1 * \ddot{\theta}_2, \dot{\theta}_2 * \ddot{\theta}_1, \dot{\theta}_2 * \ddot{\theta}_2$) are also not relevant to the dynamics. The ridge regression parameter for each of these terms becomes very large in the parameter optimization. The effect of this is to force the estimated parameter β_i for these terms to be zero and the terms to have no effect on the regression.

We have also explored stepwise regression procedures to determine which terms of the local model are useful (Draper and Smith 1981) with similar results.

7 Tuning Architectural Parameters Locally

In the process of implementing various robot learning algorithms it has become clear to us that the architectural parameters should depend on the location of the query point. In this section we describe new procedures that estimate local values of the fit parameters optimized for the site of the current query point. We want to demonstrate the differences between local and global fitting in an example where we only focus on the kernel width k of a Gaussian weighting function. In Figure 5a, a noisy data set of the function $y = x - \sin^3(2\pi x^3) \cos(2\pi x^3) \exp(x^4)$ was fitted by locally weighted regression with a globally optimized constant k . In the left half of the plot, the regression starts to fit noise because k had to be rather small to fit the high frequency regions on the right half of the plot. The prediction intervals, which will be introduced below, demonstrate high uncertainty in several places. To avoid such undesirable behavior, a local optimization criterion is needed. Standard linear regression analysis provides a series of well-defined statistical tools to assess the quality of fits, such as coefficients of determination, t-tests, F-test, the PRESS-statistic, Mallows' Cp-test, confidence intervals, prediction intervals, and many more (e.g. Myers 1990). These tools can be adapted to locally weighted regression. We do not want to discuss all possible available statistics here but rather focus on two that have proved to be quite helpful.

Cross validation has a relative in linear regression analysis called the PRESS residual error. The PRESS statistic performs leave-one-out cross validation, however, without the need to recalculate the regression parameters for every excluded point. This is computationally very efficient. The PRESS residual can be expressed as a mean squared cross validation error MSE_{cross} . In Figure 5b, the same data as in Figure 5a was fitted by adjusting k to minimize MSE_{cross} at each query point. The outcome is much smoother than that of global cross validation, and also the prediction intervals are narrower. It should be noted that the extrapolation properties on both sides of the graph are quite appropriate (compared to the known underlying function), in comparison to Figure 5a and Figure 5c.

Prediction intervals I_i are expected bounds of the prediction error at a query point x_i (Myers 1990). Besides using the intervals to assess the confidence in the fit at a certain point, they provide another optimization measure. Figure 5c demonstrates the result when applying this statistic for optimization of k at each query point. Again, the fitted curve is significantly smoother than the global cross validation fit. A rather interesting and also typical effect happens at the right side of the plot. When starting to extrapolate, the prediction intervals suddenly favor a global regression instead of the local regression, i.e., the k was chosen to be rather large. It turns out that in local optimization one always finds a competition between local and global regression. But sudden jumps from one mode into the other typically take place only when the prediction intervals are so large that the data is not reliable anyway.

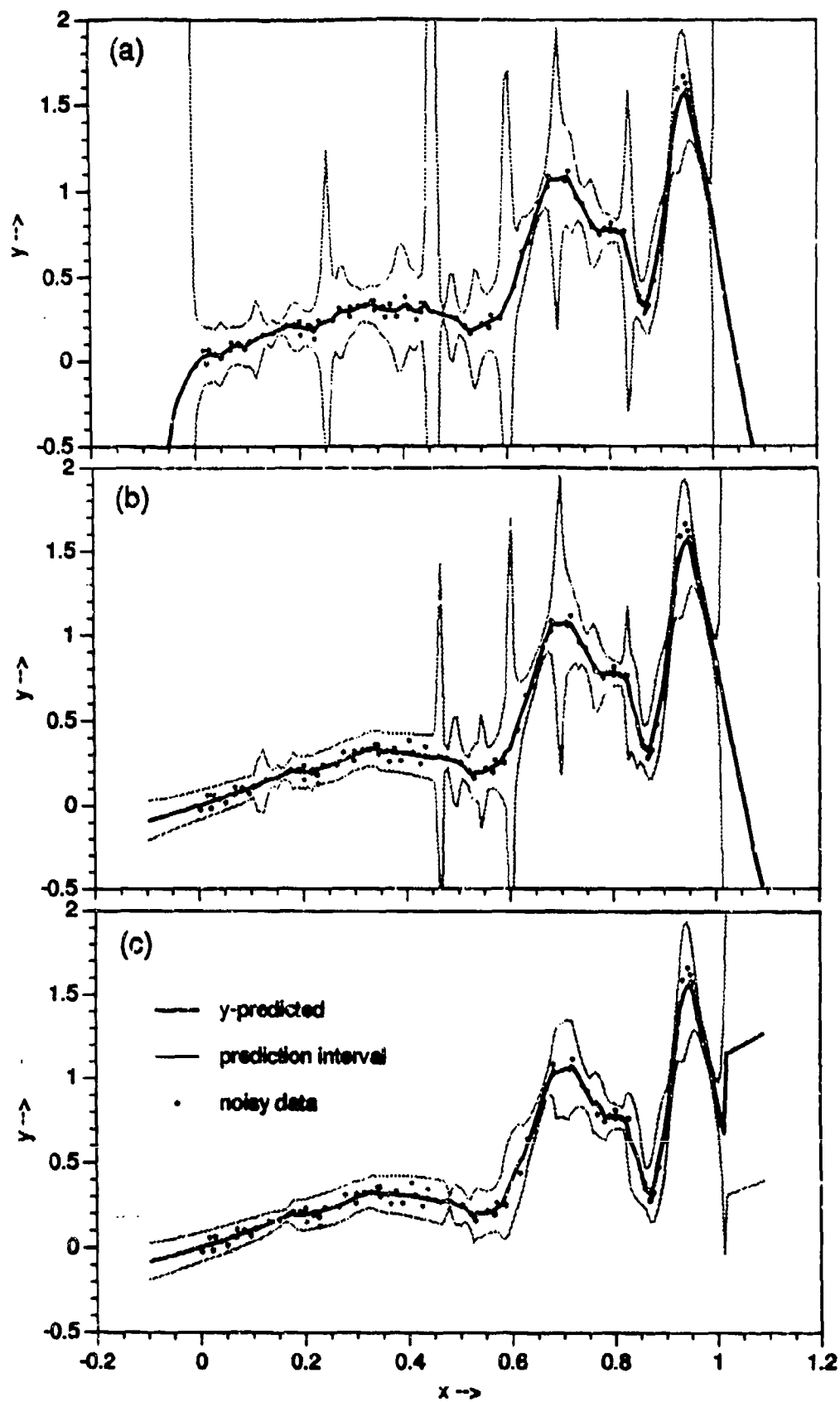


Figure 5: Optimizing the LWR fit using different measures: (a) global cross validation, (b) local cross validation, (c) local prediction intervals

8 Assessing The Quality of the Local Model

Both the local cross validation error MSE_{cross} and the prediction interval I_i may serve to assess the quality of the local fit:

$$Q_{fit} = \frac{\sqrt{MSE_{cross}}}{c} \quad (9)$$

or

$$Q_{fit} = \frac{I_i^+ - I_i^-}{c} \quad (10)$$

The factor c makes Q_{fit} dimensionless and normalizes it with respect to some user defined quantity. In our applications, we usually preferred Q_{fit} based on the prediction intervals, which is the more conservative assessment.

9 Dealing with Outliers

Linear regression is not robust with respect to outliers. This also holds for locally weighted regression, although the influence of outliers will not be noticed unless the outliers lie close enough to a query point. In Figure 6a we added three outliers to the test data of Figure 5 to demonstrate this effect; the plots in Figure 6 should be compared to Figure 5c. Moore and Atkeson (1993) applied the median absolute deviation procedure from robust statistics (Hampbell et al., 1985) to globally remove outliers in LWR. Again, we would like to localize our criterion for outlier removal. The PRESS statistic can be modified to serve as an outlier detector in LWR. For this, we need the standardized individual PRESS residual. This measure has zero mean and unit variance. If, for a given data point it deviates from zero more than a certain threshold, the point can be called an outlier. A conservative threshold would be 1.96, discarding all points lying outside the 95% area of the normal distribution. In our applications, we used 2.57, cutting off all data outside the 99% area of the normal distribution. As can be seen in Figure 6b, the effects of the outliers are reduced.

10 A Testbed for Learning Algorithms: Robot Juggling

We have constructed a system for experiments in real-time motor learning (Van Zyl 1991). The task is a juggling task known as "devil sticking". A center stick is batted back and forth between two handsticks (Figure 7A). Figure 7B shows a sketch of our devil sticking robot. The juggling robot uses its top two joints to perform planar devil sticking. Hand sticks are mounted on the robot with springs and dampers. This implements a passive catch. The center stick does not bounce when it hits the hand stick, and therefore requires an active throwing motion by the robot. To simplify the problem the center stick is constrained by a boom to move on the surface of a sphere. For small movements the center stick movements are approximately planar. The boom also provides a way to measure the current state of the center stick. Ultimately we want to be able to perform unconstrained three dimensional devil sticking using vision to provide sensing of the center stick state.

The task state is the predicted location of where the center stick would hit the hand stick if the hand stick was held in a nominal position. Standard ballistics equations for the flight of the center stick are used to map flight trajectory measurements $(x(t), y(t), \theta(t))$ into a task state:

$$\mathbf{x} = (p, \theta, \dot{z}, \dot{y}, \dot{\theta}) \quad (11)$$

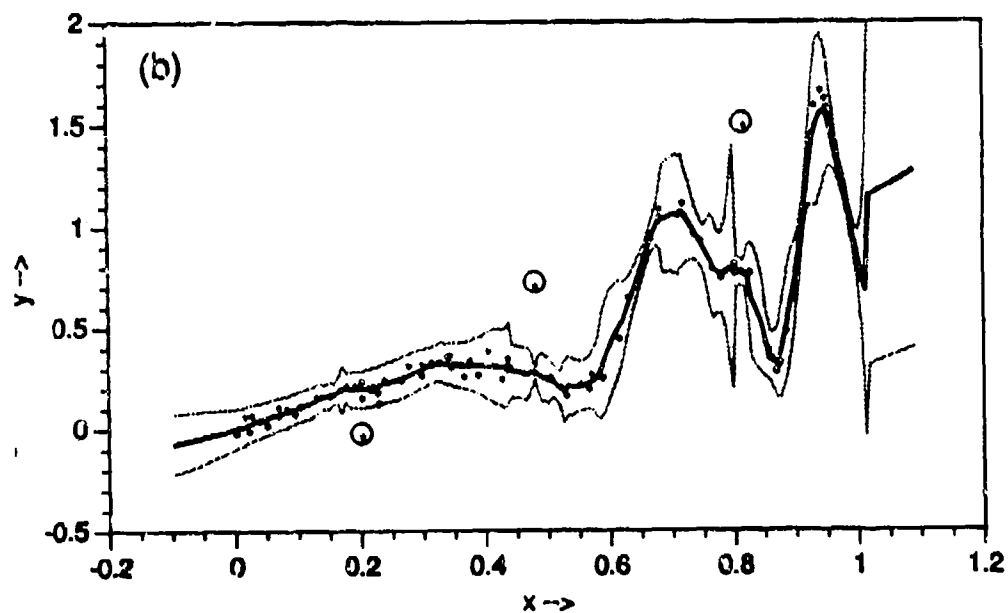
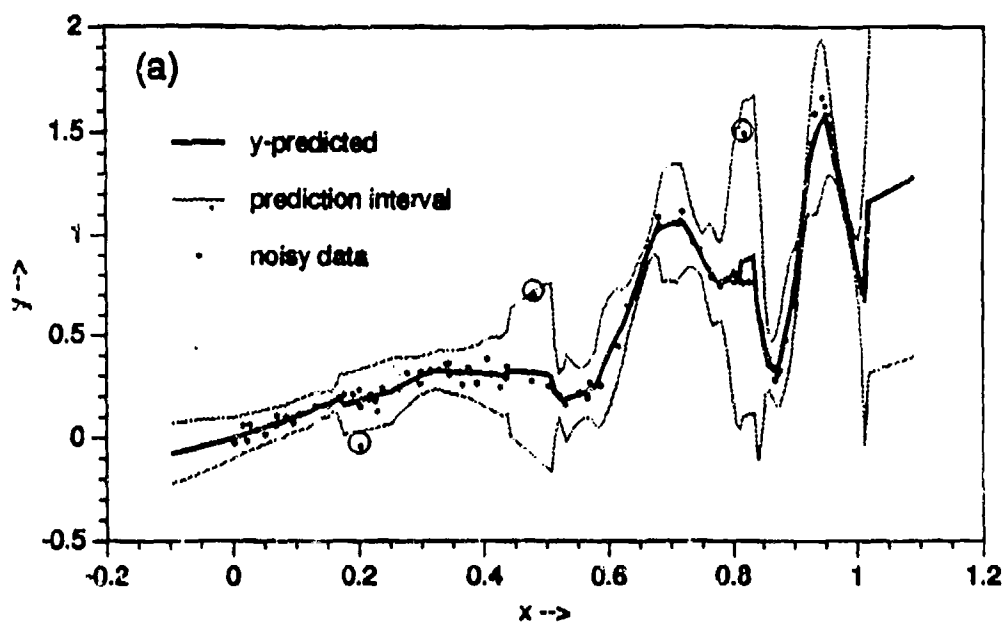


Figure 6: Influence of outliers on LWR: (a) no outlier removal, (b) with outlier removal.

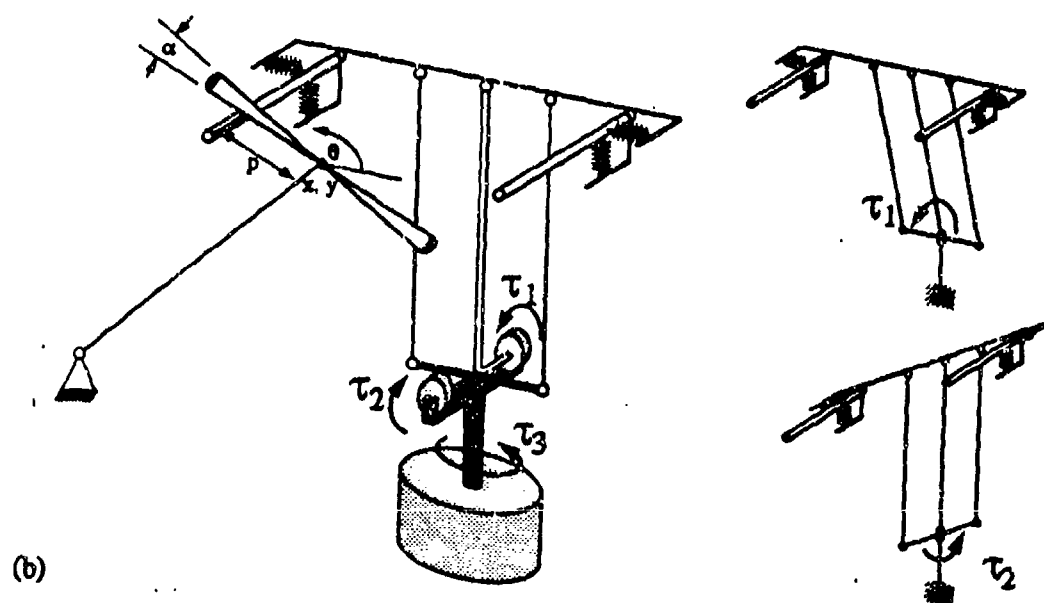
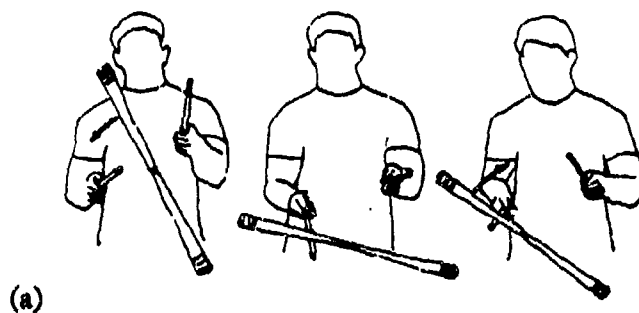


Figure 7: (a) an illustration of devil sticking, (b) a devil sticking robot.

p is the distance from the middle of the center stick that the hand stick at the nominal position contacts the center stick, θ is the angle of the center stick at nominal contact, and $\dot{x}$, $\dot{y}$, and $\dot{\theta}$, are the velocities and angular velocity of the center stick at nominal contact.

The task command is given by a displacement of the hand stick from the nominal position (x_h, y_h) , a center stick angular velocity threshold $(\dot{\theta}_t)$ to trigger the start of a throwing motion, and a throw velocity vector (v_x, v_y) .

$$\mathbf{u} = (x_h, y_h, \dot{\theta}_t, v_x, v_y) \quad (12)$$

Every time the robot catches and throws the devil stick it generates an experience of the form $(\mathbf{x}_k, \mathbf{u}_k, \mathbf{x}_{k+1})$ where $\mathbf{x}_k$ is the current state, $\mathbf{u}_k$ is the action performed by the robot, and $\mathbf{x}_{k+1}$ is the state of the center stick after the hit. Thus, a forward or an inverse model would have 10 input dimensions and 5 output dimensions.

Initially we explored learning an inverse model of the task, using nonlinear "deadbeat" control to attempt to eliminate all error on each hit. Each hand had its own inverse model of the form:

$$\mathbf{u}_k = \hat{f}^{-1}(\mathbf{x}_k, \mathbf{x}_{k+1}) \quad (13)$$

Before each hit the system looked up a command with the predicted nominal impact state and the desired result state:

$$\mathbf{u}_k = \hat{f}^{-1}(\mathbf{x}_k, \mathbf{x}_d) \quad (14)$$

Inverse model learning was successfully used to train the system to perform the devil sticking task. Juggling runs up to 100 hits were achieved. The system incorporated new data in real time, and used databases of several hundred hits. Lookups took less than 20 milliseconds, and therefore several lookups could be performed before the end of the flight of the center stick. Later queries incorporated more measurements of the flight of the center stick and therefore more accurate predictions of the state of the task. However, the system required substantial structure in the initial training to achieve this performance. The system was started with a default command that was appropriate for open loop performance of the task. Each control parameter was varied systematically to explore the space near the default command. A global linear model was made of this initial data, and a linear controller based on this model was used to generate an initial training set for the memory-based system (approximately 100 hits). Learning with small amounts of initial data was not possible.

We also experimented with learning based on both inverse and forward models. After a command is generated by the inverse model, it can be evaluated using a memory-based forward model with the same data:

$$\hat{\mathbf{x}}_{k+1} = \hat{f}(\mathbf{x}_k, \hat{\mathbf{u}}_k) \quad (15)$$

Because it produces a local linear model, the locally weighted regression procedure will produce estimates of the derivatives of the forward model with respect to the commands as part of the estimated parameter vector β . These derivatives can be used to find a correction to the command vector that reduces errors in the predicted outcome based on the forward model.

$$\frac{\partial \hat{f}}{\partial \mathbf{u}} \Delta \hat{\mathbf{u}}_k = \hat{\mathbf{x}}_{k+1} - \mathbf{x}_d \quad (16)$$

The pseudo-inverse of the matrix $\partial \hat{f} / \partial \mathbf{u}$ is used to solve the above equation for $\Delta \hat{\mathbf{u}}_k$, to handle situations in which the matrix is singular or there are a different number of commands and states (which does not apply for devil sticking). This process of command refinement can be repeated until

the forward model no longer produces accurate predictions of the outcome. This will happen when the query to the forward model requires significant extrapolation from the current database. The distance to the nearest stored data point can be used as a crude measure of the validity of the forward model estimate.

We investigated this method for incremental learning of devil sticking in simulations. The outcome, however, did not meet expectations: without sufficient initial data around the setpoint, the algorithm did not work. We see two reasons for this. First, similar to the pure inverse model approach, the inverse-forward model acts as a one-step deadbeat controller in that it tries to eliminate all error in one time step. One-step deadbeat control applies unreasonably large commands to correct for deviations from the setpoint. The workspace bounds and command bounds of our devil sticking robot limit the size of the commands. In addition, deadbeat control in the presence of errors in the model seems to lead to large inappropriate commands. Second, the ten dimensional input space is large, and even if experiences are uniformly randomly distributed in the space there is often not enough data near a particular point to make a robust inverse or forward model.

Thus, two ingredients had to be added to the devil sticking controller. First, the controller should not be deadbeat. It should plan to attain the goal using multiple control actions. Second, the control must have as the primary goal increasing the data density in the current region of the state-action space, and as a secondary goal to arrive at the desired goal state. Both requirements are fulfilled by a simple exploration algorithm we have developed, the shifting setpoint algorithm (SSA). Applied to devil sticking, the SSA proceeds as follows:

1. Regardless of the poor juggling quality of the robot (i.e., at most two or three hits per trial), the SSA makes the robot repeat these initial actions with small random perturbations until a cloud of data was collected somewhere in state-action space of each hand. An abstract illustration for this is given in Figure 8.
2. Each point in the data cloud of each hand is used as a candidate for a setpoint of the corresponding hand by trying to predict its output from its input with locally weighted regression. The point achieving the narrowest local confidence interval becomes the setpoint of the hand and an linear quadratic (LQ) controller is calculated from its local linear model (Anderson and Moore, 1990). By means of these controllers, the amount of data around the setpoints can quickly be increased until the quality of the local models exceeds a chosen statistical threshold.
3. At this point, the setpoints are gradually shifted towards the goal setpoints until the data support of the local models falls below a statistical value. After shifting, the smoothing kernel is optimized by minimizing the local cross validation error.
4. The SSA continues by collecting data in the new regions of the workspace until the setpoints can be shifted again (Fig. 8 bottom-left). The procedure terminates by reaching the goal, leaving a (hyper-) ridge of data in space (Figure 8 bottom-right).

The linear quadratic controllers play a crucial role for devil sticking. It is difficult to build good local linear models in the high dimensional forward models, particularly at the beginning of learning. Linear quadratic control is robust even if the underlying linear models are imprecise. We tested the SSA in a noise corrupted simulation and on the real robot. Learning curves are given in Figure 9a and Figure 9b.

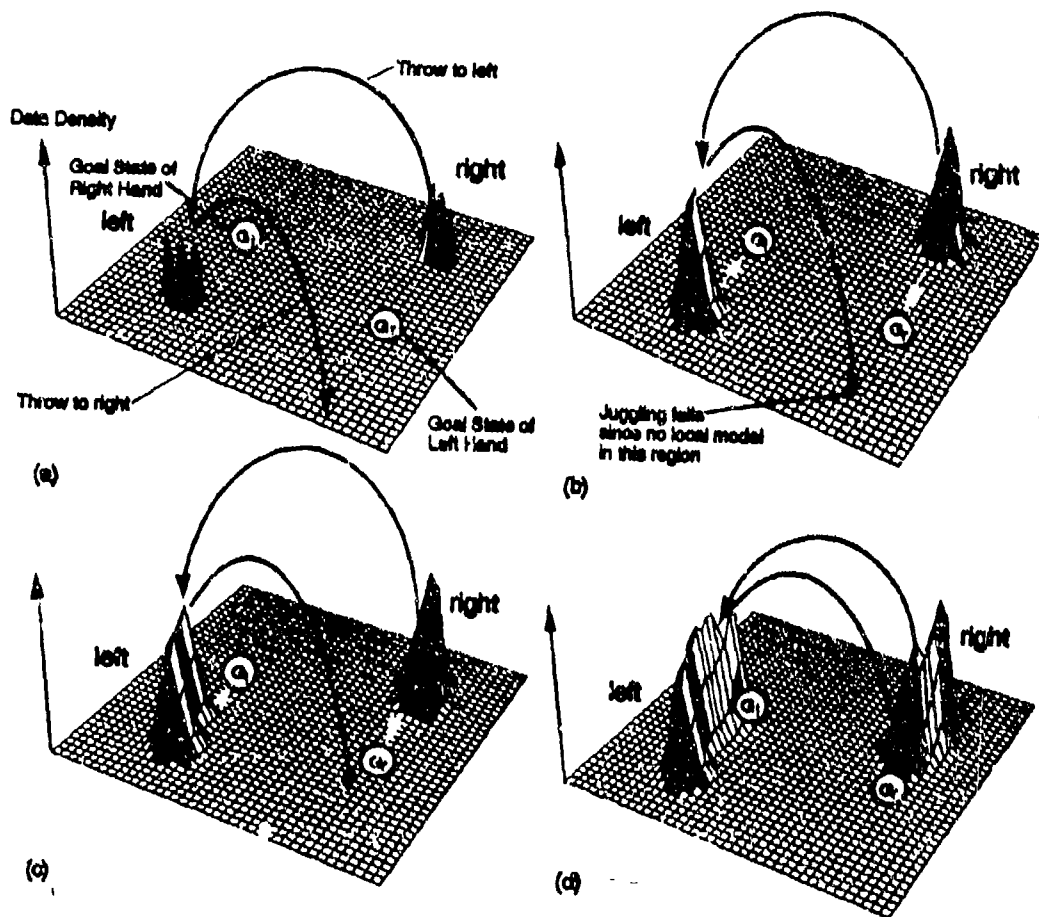


Figure 8: Abstract illustration how the SSA algorithm collects data in space: top-left: sparse data after the first few hits; top-right: high local data density due to local control in this region; bottom-left: increased data density on the way to the goals due to shifting the setpoints; bottom-right: ridge of data density after the goal was reached.

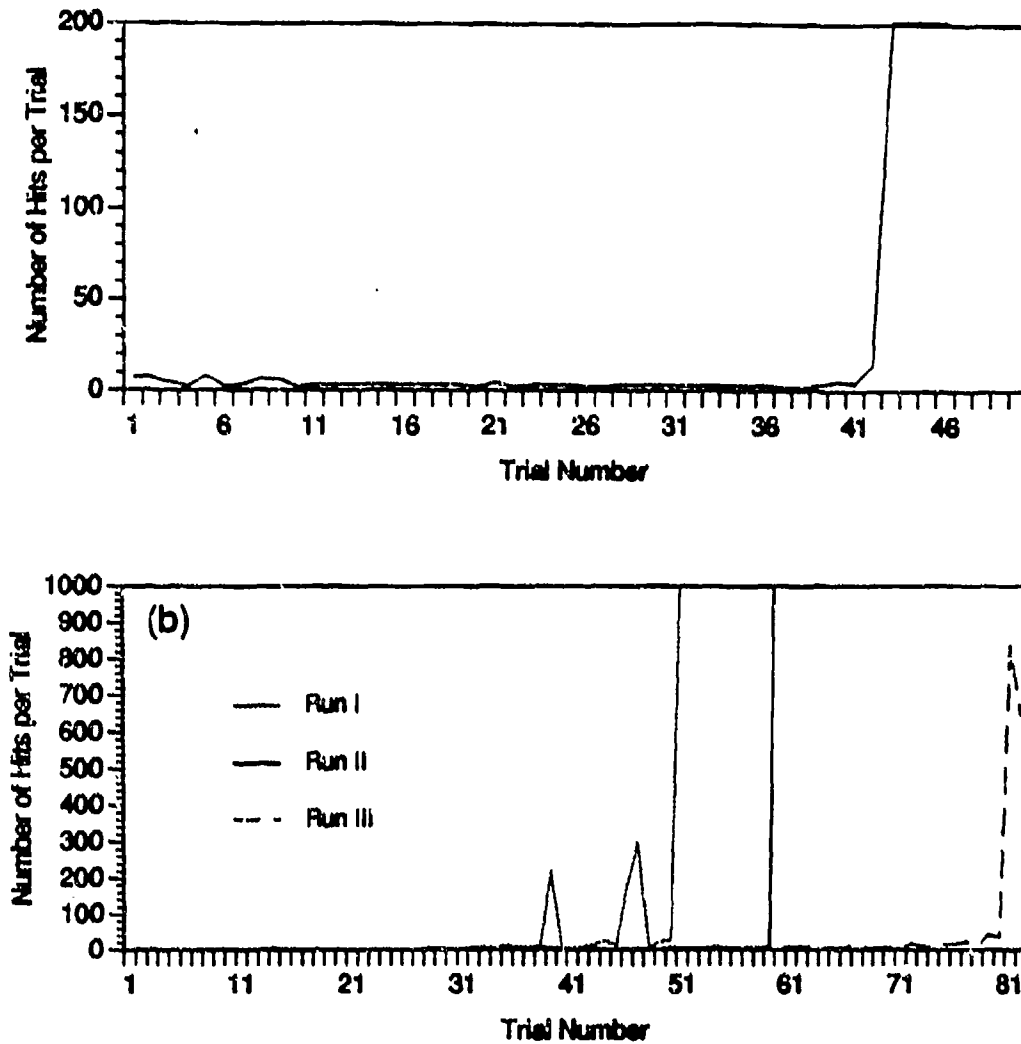


Figure 9: Learning curves of devil sticking using the SSA algorithm (a) simulation results (individual trials were stopped after 200 hits were reached), (b) real robot results.

Table 1: Comparison of parametric and memory-based approaches

	Training	Lookup	Tuning
Nonlinear Parametric Model	Nonlinear Parameter Estimation	Cheap	?
Memory- Based Model	Cheap	Linear Parameter Estimation	Nonlinear Parameter Estimation

The learning curves are typical for the given problem. It takes roughly 40 trials before the setpoint of each hand has moved close enough to the other hand's setpoint. For the simulation a break-through occurs and the robot rarely loses the devilstick after that. The real robot takes more trials to achieve longer juggling runs, and its performance is less consistent. The devil sticking robot is a very fast robot, but its positioning accuracy achieves at most ± 1 cm. Additionally, the direct drive motors do not always deliver the torque as commanded. The simulation does not model such disturbances. It only copes with various levels of Gaussian noise, which is rather well-behaved in comparison to what the real robot experiences. On average, humans need roughly a week of one hour practice a day before they learn to juggle the devilstick. With respect to this, the robot learned rather quickly. Future work will attempt to improve the learning rate and robustness; the results shown stem from very recent work.

11 Discussion

Memory-based neural networks are useful for motor learning. Fast training is achieved by modularizing the network architecture: the memory network does not need to search for weights in order to store the samples, and local models can be linear in the unknown parameters, leading to a single optimum which can be found by linear regression or gradient descent. The combination of storing all the data and only using a certain number of nearby samples to form a local model minimizes interference between old and new data, and allows the range of generalization to depend on the density of the samples.

It is useful to compare memory-based function approximation and other nonlinear parametric modeling approaches (Table 1). Training a memory-based model is computationally inexpensive, as the data is simply stored in a memory. Training a nonlinear parametric model typically requires an iterative search for the appropriate parameters. Examples of iterative search are the various gradient descent techniques used to train neural network models. Lookup or evaluating a memory-based model is computationally expensive, as described in this paper. Lookup in a nonlinear parametric model is often relatively inexpensive. If there is a situation in which a fixed set of training data is

available, and there will be many queries to the model after the training data is processed, then it makes sense to use a nonlinear parametric model. However, if there is a continuous stream of new training data intermixed with queries, as there typically is in many motor learning problems, it may be less expensive to train and query a memory-based model than it is to train and query a nonlinear parametric model.

A potential disadvantage of the memory-based approach is the limited capacity of the memory network. In this version of the proposed neural network architecture, every experience is stored. Eventually all the memory units will be used up. We have not yet needed to address this issue in our experiments. However, we plan to explore how memory use can be minimized based on several approaches. One approach is to only store "surprises". The system would try to predict the outputs of a data point before trying to store it. If the prediction is good, it is not necessary to store the point. Another approach is to forget data points. Points can be forgotten or removed from the database based on age, proximity to queries, or other criteria. It is an empirical question as to how large a memory capacity is necessary for this network design to be useful. Because memory-based learning retains the original training data, forgetting can be explicitly controlled.

The cross validation approach to optimizing the fit parameters reduces the number of arbitrary choices that need to be made before the training data is collected. However, like other modeling approaches, the choice of representation of the data (number and selection of dimensions to be measured, etc.) play a large role in determining the success of the approach.

In this learning paradigm the feedback controller serves as the teacher, or source of new data for the network. If the feedback controller is of poor quality, the nearest neighbor function approximation method tends to get "stuck" with a non-zero error level. The use of a local model seems to eliminate this stuck state, and reduce the dependence on the quality of the feedback controller.

Much work remains ahead in developing new learning paradigms. We need to develop learning systems that maintain multiple levels of models, allowing generalization via abstract models of the task. We need paradigms that are capable of finding new strategies for a task, and learning and generalizing across multiple tasks. We look forward to paradigms that perform qualitative physical reasoning and guide learning using this information. Finally, careful control of exploration is needed for improvements in learning efficiency.

Acknowledgments

B. Widrow and J. D. Cowan made the author aware of the work of Steinbuch and Taylor (Steinbuch and Widrow 1965, Cowan and Sharp 1988).

Support was provided under Air Force Office of Scientific Research grant F49-6209410362, and by the ATR Human Information Processing Research Laboratories. Support for Stefan Schaal was provided by the German Scholarship Foundation and the Alexander von Humboldt Foundation. Support for Christopher Atkeson was provided by a National Science Foundation Presidential Young Investigator Award. We thank Gideon Stein for implementing the first version of LWR on the i860 microprocessor, and Gerrie van Zyl for building the devil stick robot and implementing the first version of devil stick learning.

References

Aboaf, E.W., C.G. Atkeson, and D.J. Reinkensmeyer (1988) "Task-Level Robot Learning", *Proceedings, IEEE International Conference on Robotics and Automation*, April 24-29,

1988, Philadelphia, Pennsylvania.

Aboaf, E.W., S.M. Drucker, and C.G. Atkeson (1989) "Task-Level Robot Learning: Juggling a Tennis Ball More Accurately", *Proceedings, IEEE International Conference on Robotics and Automation*, May 14-19, 1989, Scottsdale, Arizona.

Albus, J.S. (1975a) "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)." *ASME J. Dynamic Systems, Meas., Control*, 97(September):220-227.

Albus, J.S. (1975b) "Data Storage in the Cerebellar Model Articulation Controller (CMAC)." *ASME J. Dynamic Systems, Meas., Control*, 97(September):228-233.

An, C.H., C.G. Atkeson, and J.M. Hollerbach (1988) *Model-Based Control of a Robot Manipulator* MIT Press, Cambridge, MA.

Anderson, B.D.O. and J.B. Moore (1990) *Optimal Control: Linear Quadratic Methods* Prentice Hall, Englewood Cliffs, NJ.

Atkeson, C.G. (1990) "Memory-Based Approaches to Approximating Continuous Functions", *Proceedings, Workshop on Nonlinear Modeling and Forecasting*, September 17-21, 1990, Santa Fe, New Mexico. Reprinted in *Nonlinear Modeling and Forecasting*, edited by M. Casdagli and S. Eubank, 521-522. Santa Fe Institute Studies in the Sciences of Complexity, Proc. Vol. XII. Reading, MA: Addison-Wesley, 1992.

Barnhill, R.E. (1977) "Representation And Approximation of Surfaces", in *Mathematical Software III*, J.R. Rice, ed. Academic Press, New York, pp. 69-120.

Barnhill, R.E., R.P. Dube, and F.F. Little (1983) "Properties of Shepard's Surfaces", *Rocky Mountain Journal of Mathematics* 13(2): 365-382.

Belsley, D. A., E. Kuh, and R. E. Welsch (1980) *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*, John Wiley & Sons, New York.

Bitmead, R.R., M. Gevers, and V. Wertz (1990) *Adaptive Optimal Control*, Prentice Hall, Englewood Cliffs, New Jersey.

Cheng, P.E. (1984) "Strong Consistency of Nearest Neighbor Regression Function Estimators", *Journal of Multivariate Analysis*, 15, 63-72.

Cleveland, W.S. (1979) "Robust Locally Weighted Regression and Smoothing Scatterplots", *Journal of the American Statistical Association* 74, 829-836.

Cleveland, W.S. and S.J. Devlin (1988) "Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting", *Journal of the American Statistical Association* 83, 596-610.

Cleveland, W.S., S.J. Devlin and E. Grosse (1988) "Regression by Local Fitting: Methods, Properties, and Computational Algorithms", *Journal of Econometrics* 37, 87-114.

Cover, T.M. (1968) "Estimation by the Nearest Neighbor Rule", *IEEE Transactions on Information Theory*, IT-14, 50-55.

Cowan, J.D. and D.H. Sharp (1988), "Neural Nets", *Quarterly Reviews of Biophysics*, 21(3):365-427.

Crain, I.K. and B.K. Bhattacharyya (1967) "Treatment of nonequispaced two dimensional data with a digital computer", *Geos exploration*, 5:173-194.

Dennis, J.E., D.M. Gay, and R.E. Welsch (1981) "An Adaptive Nonlinear Least-Squares Algorithm", *ACM Transactions on Mathematical Software*, 7(3) 1981.

Devroye, L.P. (1978) "The Uniform Convergence of Nearest Neighbor Regression Function Estimators and Their Application in Optimization", *IEEE Transactions on Information Theory*, IT-24, 142-151.

- Devroye, L.P. (1981)** "On the Almost Everywhere Convergence of Nonparametric Regression Function Estimates", *The Annals of Statistics*, 9(6):1310-1319.
- Draper, N.R. and H. Smith (1981)** *Applied Regression Analysis*, 2nd ed., John Wiley, New York.
- Durbin, R. and D.E. Rumelhart (1989)** "Product units: a computationally powerful and biologically plausible extension to backpropagation networks." *Neural Computation*, 1:133.
- Eubank, R.L. (1988)** *Spline Smoothing and Nonparametric Regression*, Marcel Dekker, New York, pp. 384-387.
- Falconer, K.J. (1971)** "A general purpose algorithm for contouring over scattered data points", Nat. Phys. Lab. Report NAC 6.
- Farmer, J.D., and J.J. Sidorowich (1987)** "Predicting Chaotic Time Series," *Physical Review Letters*, 59(8): 845-848.
- Farmer, J.D., and J.J. Sidorowich (1988a)** "Exploiting Chaos to Predict the Future and Reduce Noise." Technical Report LA-UR-88-901, Los Alamos National Laboratory, Los Alamos, New Mexico.
- Farmer, J.D., and J.J. Sidorowich (1988b)** "Predicting Chaotic Dynamics." in *Dynamic Patterns in Complex Systems*, J.A.S. Kelso, A.J. Mandell, and M.F. Schlesinger, (eds.) World Scientific, New Jersey, pp. 265-292.
- Farwig, R. (1987)** "Multivariate Interpolation of Scattered Data by Moving Least Squares Methods", in J.C. Mason and M.G. Cox (eds), *Algorithms for Approximation*, Clarendon Press, Oxford, pp. 193-211.
- Fix, E. and J.L. Hodges, Jr. (1951)** "Discriminatory analysis, Nonparametric regression: consistency properties", Project 21-49-004, Report No. 4. USAF School of Aviation Medicine Randolph Field, Texas. Contract AF-41-(128)-31, Feb. 1951
- Fix, E. and J.L. Hodges, Jr. (1952)** "Discriminatory analysis: small sample performance", Project 21-49-004, Rep. 11 USAF School of Aviation Medicine Randolph Field, Texas. Aug. 1952.
- Franke, R. (1982)** "Scattered Data Interpolation: Tests of Some Methods." *Mathematics of Computation*, 38(157):181-200.
- Franke, R. and G. Nielson (1980)** "Smooth Interpolation of Large Sets of Scattered Data", *International Journal Numerical Methods Engineering*, 15:1691-1704.
- Friedman, J.H., J.L. Bentley, and R.A. Finkel (1977)** "An Algorithm for Finding Best Matches in Logarithmic Expected Time." *ACM Trans. on Mathematical Software*, 3(3), Sept., pp. 209-226.
- Goldberg, K.Y. and B. Pearlmutter (1988)** "Using a Neural Network to Learn the Dynamics of the CMU Direct-Drive Arm II", Technical Report CMU-CS-88-160, Carnegie-Mellon University, Pittsburgh, PA, August 1988.
- Gordon, W.J. and J.A. Wixom (1978)** "Shepard's Method of Metric Interpolation to Bivariate and Multivariate Interpolation", *Mathematics of Computation*, 32(141):253-264.
- Grosse, E. (1989)** "LOESS: Multivariate Smoothing by Moving Least Squares" in C.K. Chui, L.L. Schumaker, and J.D. Ward (eds.), *Approximation Theory VI* pp. 1-4. Academic Press, Boston.
- Hampbell, F., P. Rousseeuw, E. Ronchetti, W. Stahel (1995)** *Robust Statistics*, Wiley International.
- Hillis, D. (1985)**, *The Connection Machine*, MIT Press, Cambridge, Mass.
- Isermann, R., K.-H. Lachmann, D. Matko, R. K.-H. Drago (1992)** *Adaptive Control Systems*, Prentice Hall, New York, New York.

- Jacobson, D.H. and D.Q. Mayne (1970) *Differential Dynamic Programming* American Elsevier Publishing Company, New York.
- Jordan, M.I., and Rosenbaum, D.A. (1988) "Action" (Tech. Rep. 88-26). Computer and Information Science, University of Massachusetts, Amherst.
- Kawamura, S. and M. Nakagawa (1990) Proceedings, IEEE International Workshop on Intelligent Robots and Systems, IROS '90
- Kawamura, S., H. Noborio, and M. Nakagawa (1990) IEEE International Workshop on Intelligent Motion Control, Bogazici University, Istanbul, 20-22 August 1990.
- Kazmierczak, H. and K. Steinbuch (1963) "Adaptive Systems in Pattern Recognition," *IEEE Trans. on Electronic Computers*, EC-12, December, pp. 822-835.
- Lancaster, P. (1979) "Moving Weighted Least-Squares Methods", in B.N. Sahney (ed.), *Polynomial and Spline Approximation*, 103-120. D. Reidel Publishing, Boston.
- Lancaster, P. and K. Šalkauskas (1981) "Surfaces Generated by Moving Least Squares Methods", *Mathematics of Computation*, 37(155):141-158.
- Lancaster, P. and K. Šalkauskas (1986) *Curve And Surface Fitting* Academic Press, New York.
- Legg M.P.C. and R.P. Brent (1969) "Automatic Contouring," *Proc. 4th Australian Computer Conference*, 467-468.
- Li, K.C. (1984) "Consistency for Cross-Validated Nearest Neighbor Estimates in Nonparametric Regression", *The Annals of Statistics*, 12:230-240.
- Lodwick, G.D., and J. Whittle (1970) "A technique for automatic contouring field survey data", *Australian Computer Journal*, 2:104-109.
- Macauley, F.R. (1931) *The Smoothing of Time Series*, National Bureau of Economic Research, New York.
- McIntyre, D.B., D.D. Pollard, and Roger Smith (1968) "Computer Programs For Automatic Contouring", *Kansas Geological Survey Computer Contributions 23*, University of Kansas, Lawrence, Kansas.
- McLain, D.H. (1974) "Drawing Contours From Arbitrary Data Points", *The Computer Journal*, 17(4):318-324.
- Miller, W.T., F.H. Glanz, and L.G. Kraft (1987) "Application of a general learning algorithm to the control of robotic manipulators", *International Journal of Robotics Research*, 6:84-98.
- Moore, A.W. and C.G. Atkeson (1993) "An Investigation of Memory-Based Function Approximators for Learning Control," Manuscript in preparation.
- More, J.J., B.S. Garbow, and K.E. Hillstrom (1980) "User Guide for MINPACK-1", ANL-80-74, Argonne National Laboratory, Argonne, Illinois.
- Müller, H.G. (1987) "Weighted Local Regression and Kernel Methods for Nonparametric Curve Fitting", *Journal of the American Statistical Association*, 82:231-238.
- Myers, R.H. (1990) *Classical and Modern Regression With Applications*, PWS-KENT, Boston, Massachusetts.
- Narendra, K.S., R. Ortega, and P. Dorato eds. (1991) *Advances in Adaptive Control* IEEE Press, Piscataway, New Jersey.
- Palmer, J.A.B. (1969) "Automated mapping," *Proc. 4th Australian Computer Conference*, 463-466.
- Pelto, C.R., T.A. Elkins, and H.A. Boyd (1968) "Automatic contouring of irregularly spaced data", *Geophysics*, 33:424-430.

- M. J. D. Powell (1987)** "Radial basis functions for multivariable interpolation: A review". In J. C. Mason and M. G. Cox (ed.), *Algorithms for Approximation*, 143-167. Clarendon Press, Oxford.
- Royall, R.M. (1966)** "A class of nonparametric estimators of a smooth regression function", Ph. D. dissertation and Tech Report No. 14, Public Health Service Grant USPHS-5T1 GM 25-09, Department of Statistics, Stanford University.
- Sabin, M.A. (1980)** "Contouring - A Review of Methods for Scattered Data", in *Mathematical Methods in Computer Graphics and Design*, K.W. Brodlie (ed.), Academic Press, New York. pp. 63-86.
- Schagen, I.P. (1984)** "Sequential Exploration of Unknown Multi-dimensional Functions as an Aid to Optimization", *IMA Journal of Numerical Analysis* 4:337-347.
- Seber, G.A.F. (1977)** *Linear Regression Analysis*, John Wiley, New York.
- Shepard, D. (1968)** "A two-dimensional function for irregularly spaced data", *Proceedings of 23rd ACM National Conference*, 517-524.
- Sheppard, W. F. (1912)** "Reduction of Errors by Means of Negligible Differences," *Proceedings of the Fifth International Congress of Mathematicians*, Vol II, pp. 348-384, E. W. Hobson and A. E. H. Love (eds), Cambridge University Press.
- Sheriff, C. W. M. (1920)** "On a Class of Graduation Formulae," *Proceedings of the Royal Society of Edinburgh*, XL: 112-128.
- Slotine, J.J., and W. Li (1991)** *Applied Nonlinear Control*, Prentice Hall, Englewood Cliffs, New Jersey.
- Specht, D.E. (1991)** "A General Regression Neural Network," *IEEE Transactions on Neural Networks*, 2(6):568-576.
- Stanfill, C., and D. Waltz (1986)** "Toward Memory-Based Reasoning." *Communications of the ACM*, vol. 29 no. 12, December, pp. 1213-1228.
- Steinbuch, K (1961)** "Die lernmatrix," *Kybernetik*, 1:36-45.
- Steinbuch, K. and U. A. W. Piske (1963)** "Learning Matrices and Their Applications," *IEEE Trans. on Electronic Computers*, EC-12, December, pp. 846-862.
- Steinbuch, K. and B. Widrow (1965)** "A Critical Comparison of Two Kinds of Adaptive Classification Networks," *IEEE Trans. on Electronic Computers*, EC-14, October, pp. 737-740.
- Stone, C.J. (1975)** "Nearest Neighbor Estimators of a Nonlinear Regression Function", *Proc. of Computer Science and Statistics: 8th Annual Symposium on the Interface* pp. 413-418.
- Stone, C.J. (1977)** "Consistent Nonparametric Regression", *The Annals of Statistics* 5, 595-645.
- Stone, C.J. (1982)** "Optimal Global Rates of Convergence for Nonparametric Regression", *The Annals of Statistics*, 10(4):1040-1053.
- Taylor, W.K. (1959)** "Pattern Recognition By Means Of Automatic Analogue Apparatus", *Proceedings of The Institution of Electrical Engineers*, 106B:198-209.
- Taylor, W.K. (1960)** "A parallel analogue reading machine", *Control*, 3:95-99.
- Taylor, W.K. (1964)** "Cortico-thalamic organization and memory", *Proc. Royal Society B*, 159:466-478.
- Van Zyl, G. (1991)** "Design and control of a robotic platform for machine learning", MS thesis, MIT Dept. of Mechanical Engineering.
- Walters, R.F. (1969)** "Contouring by Machine: A User's Guide", *American Association of Petroleum Geologists Bulletin* 53(11):2324-2340.
- Waltz, D.L. (1987)** "Applications of the Connection Machine", *Computer*, 20(1), 85-97, January.

- Watson, G.S. (1964)** "Smooth Regression Analysis", *Sankhyā: The Indian Journal of Statistics, Series A*, 26:359-372.
- White, D.A. and D.A. Sofge (1992)** *Handbook of Intelligent Control: Neural Fuzzy, and Adaptive Approaches*, Van Nostrand Reinhold, New York, New York.
- Whittaker, E., and G. Robinson (1924)** *The Calculus of Observations*, Blackie & Son, London.

The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces

Andrew W. Moore

Carnegie Mellon University
School of Computer Science &
Robotics Institute
5000 Forbes Ave
Pittsburgh, PA 15213
awm@cs.cmu.edu

Christopher G. Atkeson

Georgia Institute of Technology
College of Computing
801 Atlantic Drive
Atlanta, GA, 30332
cga@cc.gatech.edu

January 24, 1994

ABSTRACT

Parti-game is a new algorithm for learning from delayed rewards in high dimensional continuous state-spaces. In high dimensions it is essential that learning does not explore or plan over state-space uniformly. Parti-game maintains a decision-tree partitioning of state-space and applies techniques from game-theory and computational geometry to efficiently and adaptively concentrate high resolution only on critical areas. The current version of the algorithm is designed to find feasible solutions to high dimensional problems. Future versions will be designed to find a solution that optimizes a real-valued criterion. Many simulated problems have been tested, ranging from two-dimensional to nine-dimensional state-spaces, including mazes, path planning, non-linear dynamics, and planar snake robots in restricted spaces. In all cases, a good solution is found in less than twenty trials and a few minutes.

1 Reinforcement Learning

Reinforcement learning [Michie and Chambers, 1968, Sutton, 1984, Watkins, 1989, Barto *et al.*, 1991] is a promising method for robots to program and improve themselves. This paper addresses one of reinforcement learning's biggest stumbling blocks: the curse of dimensionality [Bellman, 1957], in which costs increase exponentially with the number of state variables. These costs include both the computational effort required for planning and the physical amount of data that the control system must gather.

Much work has been performed with discrete state-spaces: in particular a class of Markov decision tasks known as grid worlds [Watkins, 1989, Sutton, 1990a]. Most potentially useful applications of reinforcement learning, however, take place in multidimensional continuous state-spaces. The obvious way to transform such state-spaces into discrete problems involves quantizing them: partitioning the state-space into a multidimensional grid, and treating each box within the grid as an atomic object. Although this can be effective (see, for instance, the pole balancing experiments of [Michie and Chambers, 1968, Barto *et al.*, 1983] which break state-space up in this way), the naive grid approach has a number of dangers which will be detailed in this paper.

This paper studies in detail the pitfalls of discretization during reinforcement learning and then introduces the parti-game algorithm. Some earlier work [Simons *et al.*, 1982, Moore, 1991, Chapman and Kaelbling, 1991, Dayan and Hinton, 1993] considered recursively partitioning state-space while learning from delayed rewards. The new ideas in the parti-game algorithm include (i) a game-theoretic splitting criterion to robustly choose spatial resolution, (ii) real-time incremental maintenance and planning with a database of previous experiences, and (iii) using local greedy controllers for high-level "funneling" actions.

2 Assumptions

The parti-game algorithm applies to learning control problems in which:

1. State and action spaces are continuous and multidimensional.
2. "Greedy" and hill-climbing techniques can become stuck, never attaining the goal.
3. Random exploration can be intractably time-consuming.
4. The system dynamics and control laws can have discontinuities and are unknown: they must be learned. However, we do assume that all paths that the system can travel through state space are continuous.

The experiments reported later all have properties 1-4. However, the initial algorithm, described and tested here, has the following restrictions:

5. Dynamics are deterministic.
6. The task is specified by a goal state, not an arbitrary reward function.
7. The goal state is known.
8. A feasible solution is found, not necessarily a path which optimizes a particular criterion.
9. A local greedy controller is available, which we can ask to move greedily towards any desired state. There is no guarantee that a request to the greedy controller will succeed. For example, in a maze a greedy path to the goal would quickly hit a wall.

This paper begins by giving a series of algorithms of increasing sophistication, culminating in parti-game. We then give results for a number of experimental domains and conclude with discussion of how constraints 5-9 may be relaxed.

3 The Parti-game Algorithm

The parti-game learns a controller from a start region to a goal region in a continuous state-space. We now give four increasingly effective algorithms which attempt to perform this by discrete partitionings of state-space. Algorithms (1) and (2) are non-learning: they plan a route to the goal given a-priori knowledge of the world. Algorithms (3) and (4) must learn, and hence explore, the world while planning a route to the goal. Algorithm (1) is a planner which assumes that state transitions begin at the center of partitions, and generalizes this to the assumption of starting randomly within a partition. Algorithm (2) avoids some of (1)'s mistakes by means of worst-case planning. Algorithm (3) is a learning version of (2). Algorithm (4) is the parti-game algorithm. It develops a variable resolution partitioning in conjunction with the planning and learning of Algorithm (3).

3.1 Algorithm (1): Non-learning and fixed partitions

A *partitioning* of a continuous state-space is a finite set of N disjoint regions labeled $1, 2, \dots, N$ such that the whole of state space is covered by the union of all partitions. Throughout this paper we will assume the partitions are all axis-aligned hyperrectangles, though this assumption is not strictly necessary. It is important to clarify a potential confusion between real-valued states and partitions. A *real-valued state*, s , is a real-valued vector in a multidimensional space. For example,

states from the maze depicted in Figure 1 are two-dimensional (x, y) coordinates. A partition is a discrete entity, and Figure 1 is broken into six partitions with identifiers 1...6. Each real-valued state is in one partition and each partition contains a continuous set of real-valued states. Define $\text{NEIGHS}(i)$ as the set of partitions which are adjacent to i . In Figure 1, $\text{NEIGHS}(1) = \{2, 4\}$.

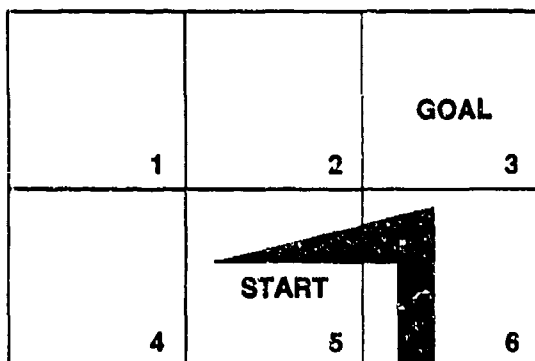


Figure 1: A two-dimensional continuous maze with one barrier: the black polygonal region near the bottom right. State-space has been discretized into six square partitions.

Algorithm (1) takes as input an environmental model and a partitioning P . The environmental model can be any model (for example, dynamic or geometric) which we can use to tell us for any real-valued state, control command and time interval, what the subsequent real-valued state will be. The algorithm outputs a *policy*: a mapping from partition identifiers to the neighboring partitions which should be aimed for. The algorithm depends upon the **NEXT-PARTITION** function, which we define first. **NEXT-PARTITION** tells us which partition we end up in if we start at a given real-valued state and keep moving toward the center of a given partition (using a local greedy controller) until we either exit our initial partition or get stuck. Let i be the partition containing real valued state s . Continue applying the local greedy controller “aim at partition j ” either until partition i is exited or until we become permanently stuck in i . Then

$$\text{NEXT-PARTITION}(s, j) = \begin{cases} i & \text{if we became stuck} \\ \text{the partition containing the exit state} & \text{otherwise} \end{cases} \quad (1)$$

The test for sticking can simply be implemented as a test to see if the system has not exited the partition after a predefined time interval. Depending upon the application other sticking detectors are possible, such as an obstacle sensor on a mobile robot.

Algorithm (1) works by constructing a discrete, deterministic Markov decision task (MDT) [Bellman, 1957, Bertsekas and Tsitsiklis, 1989] in which the discrete MDT states correspond to partitions. Actions correspond to neighbors thus: action k in partition i corresponds to starting at the center of partition i and greedily aiming at the center of partition k .

ALGORITHM (1).

- 1 Given N partitions, construct a deterministic MDT with N discrete states $1 \dots N$. The set of actions of partition i is precisely $\text{NEIGHS}(i)$. Define $\text{NEXT}(i, k)$ as

$$\text{NEXT}(i, k) = \text{NEXT-PARTITION}(\text{CENTER}(i), k) \quad (2)$$

where $\text{CENTER}(i)$ is the real-valued state at the center of partition i .

- 2 The shortest path to the goal from each partition i , denoted by $J_{SP}(i)$, is determined by solving the set of equations:

$$J_{SP}(i) = \begin{cases} 0 & \text{if } i = \text{GOAL} \\ 1 + \min_{k \in \text{NEIGHS}(i)} J_{SP}(\text{NEXT}(i, k)) & \text{Otherwise} \end{cases} \quad (3)$$

The equations are solved by a shortest-path method such as dynamic programming [Bellman, 1957, Bertsekas and Tsitsiklis, 1989] or Dijkstra's algorithm [Knuth, 1973].

- 3 The following policy is returned: Always aim for the neighbor with the lowest $J_{SP}()$ value.

This simple algorithm has immediate drawbacks. It will minimize the number of partitions to the goal, not the real distance. And the discretization can easily find impossible solutions or fail to find valid solutions. As an example of the former, in Figure 1, Algorithm (1) would find solution path $5 \rightarrow 6 \rightarrow 3$. This is because it is possible to travel from the center of 5 and enter 6 (in the part of 6 to the left of the obstacle), and it is possible to travel from the center of 6 and enter 3.

An extension to Algorithm (1) might initially appear to solve the problem. We could remove the assumption that all paths between partitions begin at the center of the source partition. Suppose we produce a stochastic Markov decision task. Let p_{ij}^k be an approximation of the probability of transition to partition j given we have started in i and aimed at the center of k . p_{ij}^k is defined by the probability we end up in partition k from a uniformly randomly chosen legal start point in partition i . The dynamic programming step of the previous algorithm is altered so that it now solves the stochastic MDT:

$$J_{SP}(i) = \begin{cases} 0 & \text{if } i = \text{GOAL} \\ 1 + \min_{k \in \text{NEIGHS}(i)} \sum_{j=1}^N p_{ij}^k J_{SP}(j) & \text{Otherwise} \end{cases} \quad (4)$$

Although intuitively appealing, this refinement does not help. In the example of Figure 1 the resultant policy from state 5 will still be to aim for 6. As we see from Figure 2, $p_{56}^6 = 0.65$, and from

Figure 3, $p_{63}^3 = 0.91$. The policy $5 \rightarrow 6 \rightarrow 3$ is interpreted as the transition graph in Figure 4 which has expected length $1/0.65 + 1/0.91 = 2.64$, and so is preferred over the longer but guaranteed policy of $5 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

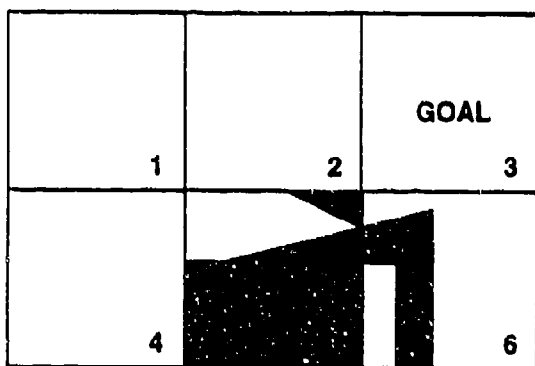


Figure 2: Approximately 65% of the starting states (those in the shaded region) in partition 5 are such that they will enter partition 6 if we aim for the center of 6. Thus $p_{66}^6 = 0.65$.

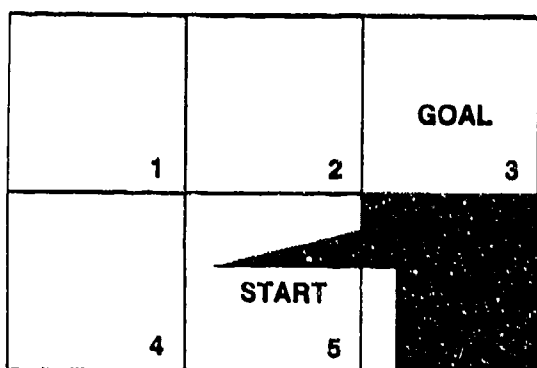


Figure 3: In a similar fashion to Figure 2, $p_{63}^3 = 0.91$.

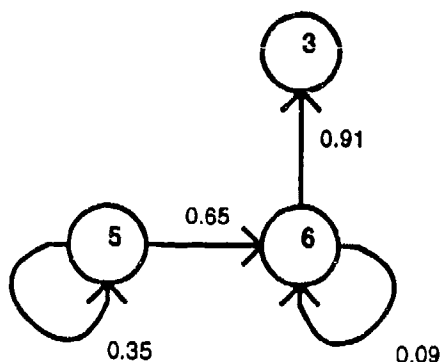


Figure 4: The partition transition probabilities if we follow the $5 \rightarrow 6 \rightarrow 3$ policy according to the assumptions in the text.

Other variants of this stochastic approximation approach are possible, but they all suffer from the same problem. They are using a Markov decision formalism for something which does not have

the Markov property. This is because from a given partition, i , the neighbors which can successfully be reached depend on more than “ i ”, they also depend on the current location within i .

3.2 Algorithm (2): Assuming the worst case

Instead of approximating the steps-to-goal value of a partition by the average steps-to-goal of all real-valued states in the partition, we approximate it by the worst value. As before, each partition has an associated set of actions, each labeled by a neighboring partition. Also, each action now has a set of possible *outcomes*. The outcomes of an action j in a partition i are defined as the set of possible next partitions.

$$\text{OUTCOMES}(i, j) = \left\{ k \mid \begin{array}{l} \text{there exists a real valued state } s \text{ in partition } i \text{ for which} \\ \text{NEXT-PARTITION}(s, j) = k \end{array} \right\} \quad (5)$$

In Figure 5, the actions are denoted by black solid arrows and the outcomes by the thin lines. For example, partition 5 has three actions: "aim at 4", "aim at 2" and "aim at 6". The "aim at 6" action has, in turn, two possible outcomes. We might make it if we are lucky, or else we will remain in 5. The **OUTCOMES()** sets are decisions which an imaginary adversary will be allowed to make, seen in Algorithm (2).

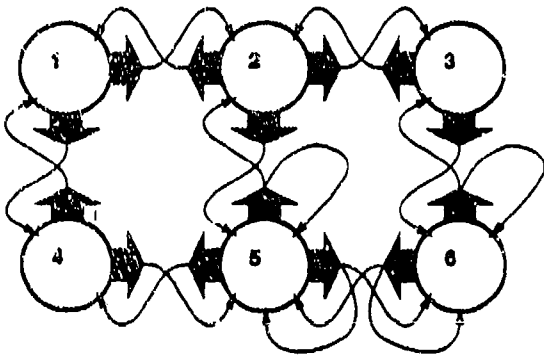


Figure 5: The symbolic representation of the original problem as partition (circles), actions (solid arrows) and outcomes (thin arrows).

ALGORITHM (2).

- 1 Define $J_{WC}(i)$ as the minimum number of partitions to the goal under the worse-case assumption that whenever we have specified our current partition i and our intended next partition j , an adversary is permitted to place us in the worst position within partition i prior to the local controller being activated.

Solve the following set of minimax equations:

$$J_{WC}(i) = \begin{cases} 0 & \text{if } i = \text{GOAL} \\ 1 + \min_{k \in \text{NEIGHS}(i)} \max_{j \in \text{OUTCOMES}(i,k)} J_{WC}(j) & \text{Otherwise} \end{cases} \quad (6)$$

where $J_{WC}(i)$ is allowed to take the value $+\infty$ to denote a partition from which our adversary can permanently prevent us reaching the goal. Call such a partition a losing partition.

- 2 The following policy is returned: Always aim for the neighbor with the lowest $J_{WC}()$ value.

The $J_{WC}(\cdot)$ function can be computed by a standard minimax algorithm [Knuth, 1973], which is in turn closely related to deterministic dynamic programming algorithms.

This algorithm is pessimistic, but if it tells us that the current partition is n partition transitions to the goal then we can be sure that if we follow its policy we will indeed take n or fewer partition transitions. The trivial inductive proof is omitted.

In Figure 1, Algorithm (2) will decide that partition 5 is four steps from the goal and will recommend heading towards 4. It avoids partition 6 because the minimax assumption scores partition 6 as being ∞ steps from the goal. This is because if, in partition 6, we decide to use action "aim for 3" the adversary will start us in the bottom left of partition 6. And if we use action "Aim for 5", the adversary will start us in the bottom right of partition 6.

It should be observed just how pessimistic the algorithm is. In the almost entirely empty maze of Figure 6 the start partition will be considered a losing partition. So although the minimax assumption guarantees success if it finds a solution, it may often prevent us from solving easy problems. We will see that Algorithm (3) reduces the severity of this problem because instead of considering the worst of all possible outcomes, the planner only considers the worst of all empirically observed outcomes. Thus a block in a piece of a partition which never was actually visited would not be identified as an outcome available to the adversary. Algorithm (4) fully solves the remaining aspects of the problem by increasing the resolution of losing partitions.

3.3 Algorithm (3): A learning version of Algorithm (2)

An important aim of this work is to have a controller which does not begin with an environmental model, but which manages instead to learn purely from experience. Algorithm (2) can be extended to permit this. The set of $\text{OUTCOMES}(i, j)$ for each partition i and neighbor j can be obtained empirically. Whenever an $\text{OUTCOMES}(i, j)$ is altered, the game is solved with the new outcomes set. We still assume that the location of the partition containing the goal is known.

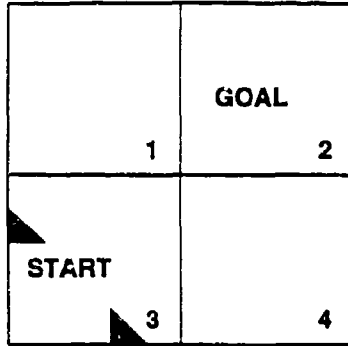


Figure 6: Partition 3 is scored as losing because if it aims for 1 the adversary can place it below the upper triangular block and if it aims for 4 the adversary can place it to the left of the lower triangular block.

A further detail must be resolved. In the early stages, what should be done for those actions which have not yet been experienced? The answer is to assume by default that any neighbor aimed for can be attained. Algorithm (3), based on these ideas, takes three inputs:

- The current real-valued system state s .
 - A partitioning of state-space, P .
 - A database, D , of all previously observed partition-transitions in the lifetime of the system.
- This is a set of triplets:

(starting in i_0 , I aimed for j_0 and actually arrived in k_0)

(starting in i_1 , I aimed for j_1 and actually arrived in k_1)

⋮

The algorithm returns two outputs: The final system state after execution and a binary signal indicating SUCCESS or FAILURE. The database is also updated according to experience.

ALGORITHM (3).

REPEAT FOREVER

- 1 Compute the $\text{OUTCOMES}(i, j)$ set for each partition i and each neighbor $j \in \text{NEIGHS}(i)$ thus:

- If, for any k , $(i, j, k) \in D$ then:

$$\text{OUTCOMES}(i, j) = \{k \mid (i, j, k) \in D\} \quad (7)$$

- Else, use the optimistic assumption in the absence of experience:

$$\text{OUTCOMES}(i, j) = \{j\} \quad (8)$$

2 Compute $J_{WC}()$ for each partition using minimax.

3 Let $i :=$ the partition containing s .

4 If $i = \text{GOAL}$ then exit, signaling SUCCESS.

5 If $J_{WC}(i) = \infty$ then exit, signaling FAILURE.

6 Else

6.1 Let $j := \underset{j' \in \text{NEIGHS}(i)}{\text{argmin}} J_{WC}(j')$.

6.2 WHILE (not stuck and s is still in partition i)

6.2.1 Actuate local controller aiming at j .

6.2.2 $s :=$ new real-valued state.

6.3 Let $i_{\text{new}} :=$ the identifier of the partition containing s .

6.4 $D := D \cup \{(i, j, i_{\text{new}})\}$

LOOP

An addition to the algorithm can reduce the computational load. If real time constraints do not permit full recomputation of J_{WC} after an outcome set has changed, then the J_{WC} updates can take place incrementally in a series of finite time intervals interleaved with real-time control decisions. Techniques like this are described in [Sutton, 1990b, Peng and Williams, 1993, Moore and Atkeson, 1993].

The following theorem has not been proved but we expect few difficulties: If a solution exists from all real-valued states in all partitions, according to Algorithm (2), then Algorithm (3) will, in fewer than N^3 partition transitions, also find a solution from its initial state, where N is the number of partitions.

A final note about Algorithm (3) is necessary. More general systems than mazes will produce more interesting games. Later we will see examples of non-uniform partitionings and of dynamics that produce curved trajectories through space. Both cases can produce more detailed game structures than stuck/non-stuck transitions, and Algorithm (3) is applicable in these cases too. Such a structure is shown in Figure 7.

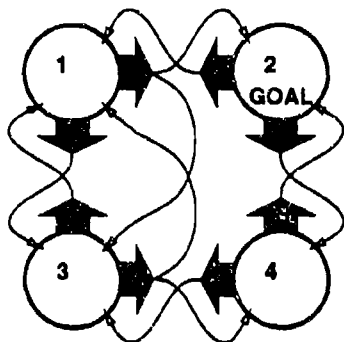


Figure 7: Partitions 1 and 3 are losers because the adversary can force a permanent loop between them.

3.4 Algorithm (4): Varying the Resolution

We do not wish the system to give up when it discovers it is in a partition for which $J_{WC} = \infty$. The correct interpretation of a losing partition is that the planner needs higher resolution, and parti-game gives it that by dividing some coarse partitions in two.

Interestingly, it is not necessarily worth increasing the resolution of the partition the system is in, nor is it necessarily worth splitting all partitions which have $J_{WC} = \infty$. Figure 8 shows a case in which the current state is in a partition which it will not help to split. This is because there is no path to a non-losing state from the current partition anyway (other losing partitions block us) and no matter how high we make its resolution our current partition will remain a loser.

LOSE	LOSE	1 STEP	GOAL
Current State	LOSE	2 STEPS	1 STEP
	LOSE	LOSE	
LOSE	LOSE		

Figure 8: A 12-cell partitioning in which it will not help to split the partition containing the current state.

It is the partitions on the border between losing regions and non-losing regions which should be

split. Under the assumption that all paths through state space are continuous, and also assuming that a path to the goal actually exists, there must currently be a hole in one of the border partitions which has been missed by the over-coarseness. This motivates the final algorithm which we present.

The algorithm takes three inputs:

- The current real-valued system state, s .
- A partitioning of state-space, P .
- A database, D , of all empirically experienced (start, aimed-for, actual-outcome) triplets.

It returns two outputs: a new partitioning of state-space and a new database.

ALGORITHM (4): (PARTIGAME).

WHILE (s not in the goal partition)

- 1 Compute $J_{WC}()$ for each partition using minimax.
- 2 Run Algorithm (3) on s and P , retrieving the resulting additions to the database D , plus the new real-valued state s , and a success/failure signal.
- 3 If FAILURE was signaled
 - 3.1 Let $Q :=$ All partitions in P for which $J_{WC} = \infty$.
 - 3.2 Let $Q' :=$ Members of Q who have any non-losing neighbors.
 - 3.3 Let $Q'' := Q'$ and all non-losing neighbors of members of Q' .
 - 3.4 Construct a new set of partitions from Q'' , of twice the size, produced by splitting each partition in half along its longest axis. Call this new set R .
 - 3.5 $P := P + R - Q''$
 - 3.6 Recompute all new neighbor relations and delete those members of database D which contain a member of Q'' as a start point, an aim-for or an actual-outcome.

LOOP

3.5 Partigame Details

Initialization

Before the very first trial, parti-game is initialized as just two partitions: a goal partition covering

the goal region, and one other large partition covering the rest of state-space. At that point, Algorithm (4) is called. Unless the system is very lucky, this trivial partitioning will not be adequate to reach the goal using the greedy controller. At the point when this is detected the initial, trivial partitioning will quickly start splitting.

Increasing the resolution

Notice that this algorithm increases the resolution at both sides of the win/lose border. This prevents enormous partitions from bordering tiny partitions. There could be other algorithms in which the partitions to split are chosen differently. The question of which alternative is best remains open for further investigation.

Planning and learning in parti-game

Partigame performs planning and learning simultaneously. Interestingly, these two components are of great help to each other. The learning consists of gathering data to build up the sets of known possible outcomes of transitions between partitions. This data is gathered by planning paths to "interesting" partitions and executing "interesting" actions. A partition seems interesting if, according to the optimistic assumption that anything we haven't tried will work (Equation 8 in Algorithm (3)), the partition lies on the shortest path to the goal.

The planning is helped by the learning because the computation and representation are concentrated on the parts of the state space which, according to the database of experiences, are most critical.

The goal partition

The goal partition is special. It never changes or gets split. The task is defined to be solved when the system enters any part of the goal partition. In the experimental diagrams in Section 4 it is the box marked "Goal".

When other partitions are split, each new partition has to recompute all the neighbors that it is next to. Any new partition which intersects the goal partition also includes the goal partition as one of its neighbors.

4 Experiments

All these experiments are broken into trials. On each trial the system is placed in an initial state and the trial proceeds until the system enters the goal region.

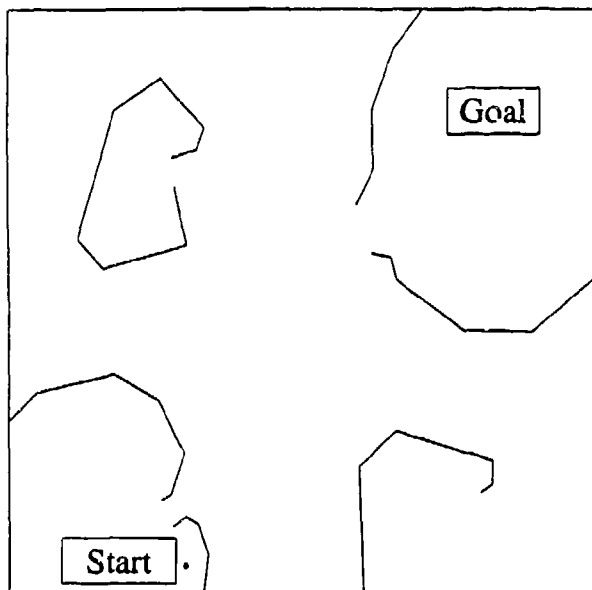


Figure 9: A two-dimensional maze problem. The point robot must find a path from start to goal without crossing any of the barrier lines. Remember that initially it does not know where any obstacles are, and must discover them by finding impassable states.

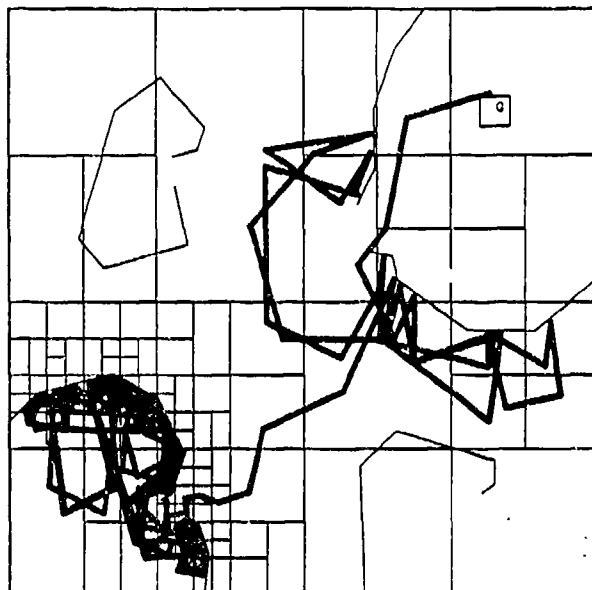


Figure 10: The path taken during the entire first trial. It begins with intense exploration to find a route out of the almost entirely enclosed start region. Having eventually reached a sufficiently high resolution, it discovers the gap and proceeds greedily towards the goal, only to be stopped by the goal's barrier region. The next barrier is traversed at a much lower resolution, mainly because the gap is larger.

4.1 Maze navigation

Figure 9 shows a two-dimensional continuous maze. Figure 10 shows the performance of the robot during the very first trial. Figure 11 shows the second trial, started from a slightly different position. The policy derived from the first trial gets us to the goal without further exploration. The trajectory has unnecessary bends. This is because the controller is discretized according to the current partitioning. If necessary, a local optimizer could be used to refine this trajectory¹.

The system does not explore unnecessary areas. The barrier in the top left remains at low resolution because the system has had no need to visit there. Figures 12 and 13 show what happens when we now start the system inside this barrier.

¹Another method is to increase the resolution along the trajectory [Moore, 1991].

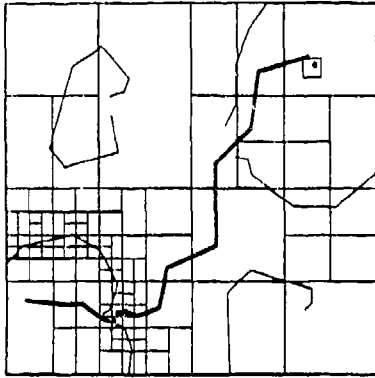


Figure 11: The second trial.

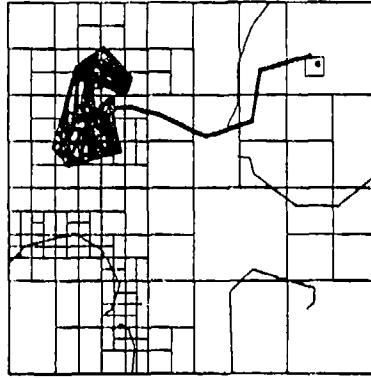


Figure 12: Starting inside the top left barrier.

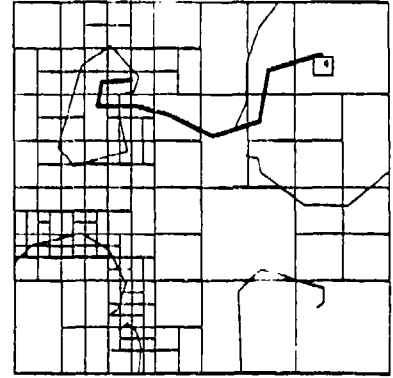


Figure 13: The trial after that.

4.2 Non-linear dynamics

Figure 14 depicts a frictionless puck on a bumpy surface. It can thrust left or right with a maximum thrust of ± 4 Newtons. Because of gravity, there is a region near the center of the hill at which the maximum rightward thrust is not strong enough to accelerate up the slope. Thus if the goal is at the top of the slope, a strategy which proceeded by greedily choosing actions to thrust towards the goal would get stuck.

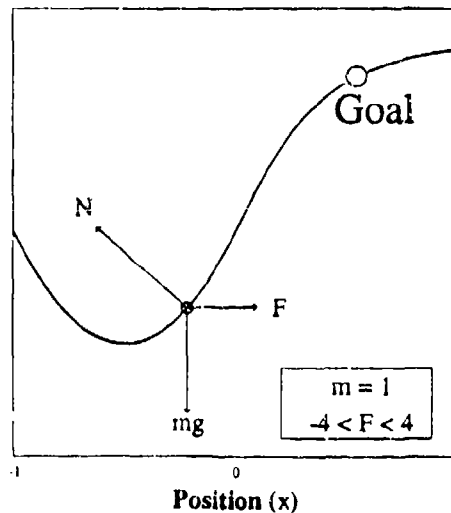


Figure 14: A frictionless puck acted on by gravity and a horizontal thruster. The puck must get to the goal as quickly as possible. There are bounds on the maximum thrust.

This is made clearer in Figure 15, a *phase space diagram*. The puck's state has two components,

the position and velocity. The hairs show the next state of the puck if it were to thrust rightwards with the maximum legal force of 4 Newtons. Notice that at the center of state space, even when this thrust is applied, the puck velocity decreases and it eventually slides leftwards. The optimal solution for the puck task, depicted in Figure 16, is to initially thrust away from the goal, gaining negative velocity, until it is on the far left of the diagram. Then it thrusts hard right, to build up sufficient energy to reach the top of the hill.

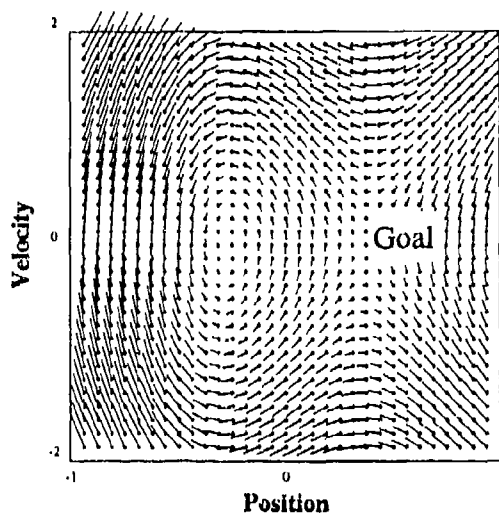


Figure 15: The state transition function for a puck which constantly thrusts right with maximum thrust.

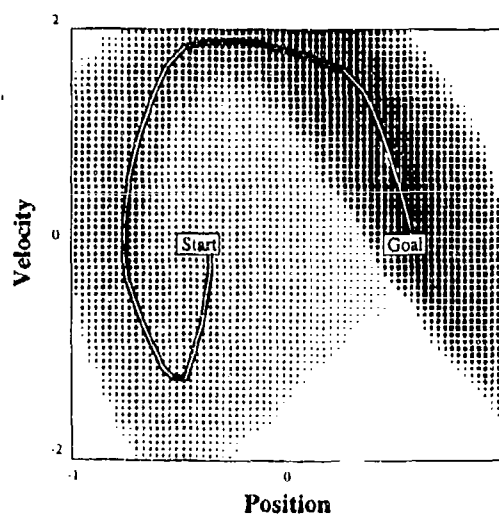


Figure 16: The "minimum-time" path from start to goal for the puck on the hill. The optimal value function is shown by the background dots. The shorter the time to goal, the larger the black dot. Notice the discontinuity at the escape velocity.

The local greedy controller which parti-game uses is bang-bang. To aim for a partition "north"

in state space—a partition with greater velocity—it thrusts with the maximum permissible force of $+4N$. To aim for a lower velocity partition it thrusts with $-4N$. To aim for an “east” or “west” partition, the local controller merely controls its velocity (using a trivial linear controller) to be equal to the velocity of the center of the destination partition. Notice that if the current partition’s velocity is greater than zero it is hopeless to greedily aim for the partition on the left. It is also hopeless to aim at the partition on the right if the current partition has negative velocity. In the experiments below Parti-game is given this extra information. Forcing parti-game to learn this from experience approximately doubles the learning time.

Figure 17 shows the trajectory through state space during the very first learning trial, while it is exploring and developing its initial partitioning. Figure 18 shows the resulting partitioning and the subsequent trajectory²; on its second trial it has already learned the basic strategy of “begin by getting a negative velocity, moving backwards, and only then heading forward with full thrust”. Figure 19 shows the interesting result of running many more trajectories, each starting at random parts of state space. Many partitions are created and refined, but only around the critical border in state space which serves as the escape velocity of the problem (also visible as the discontinuity in Figure 16). This high resolution line arises not out of any pre-programmed knowledge of the escape velocity but because the system does not need to increase the resolution of partitions which fail to intersect the escape velocity region.

4.3 Higher dimensional state spaces

Figure 20 shows a three-dimensional state-space problem. If a standard grid were used, this would need an enormous number of states because the solution requires detailed maneuvers. Parti-game’s total exploration took 18 times as much movement as one run of the final path obtained.

Figure 21 shows a four-dimensional problem in which a ball slides on a tray with steep edges. The goal is on the other side of a ridge. The maximum permissible force is low. Greedy strategies, or globally linear control rules, get stuck in limit cycles within a valley. The local greedy controller to navigate between adjacent partitions is bang-bang controller. Parti-game’s solution runs to the far end of the tray, to build up enough velocity to make it over the ridge. The exploration-length versus final-path-length ratio is 24.

Figure 22 shows a 9-joint snake-like robot manipulator which must move to a specified configuration on the other side of a barrier. Again, no kinematics model or knowledge of obstacle locations

²Careful inspection of this diagram reveals that the trajectory changes direction not at the borders of cells but instead within cells. This is because the current implementation waits until it is well within a cell before applying the cell’s recommended action.

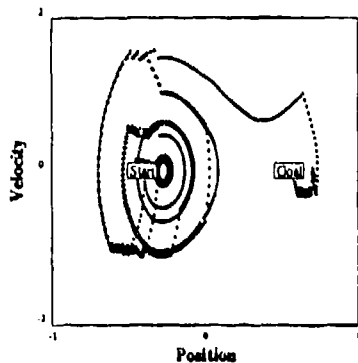


Figure 17: The trajectory of the very first trial, while the system performed its initial exploration of state space.

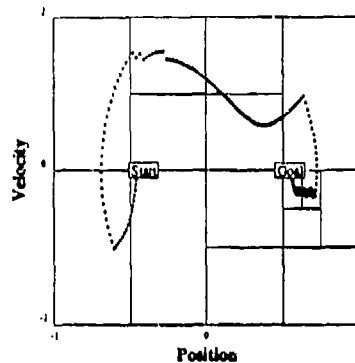


Figure 18: The trajectory and partitioning of the second trial.

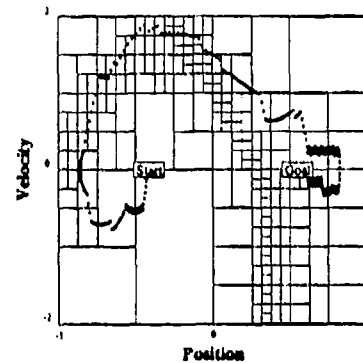


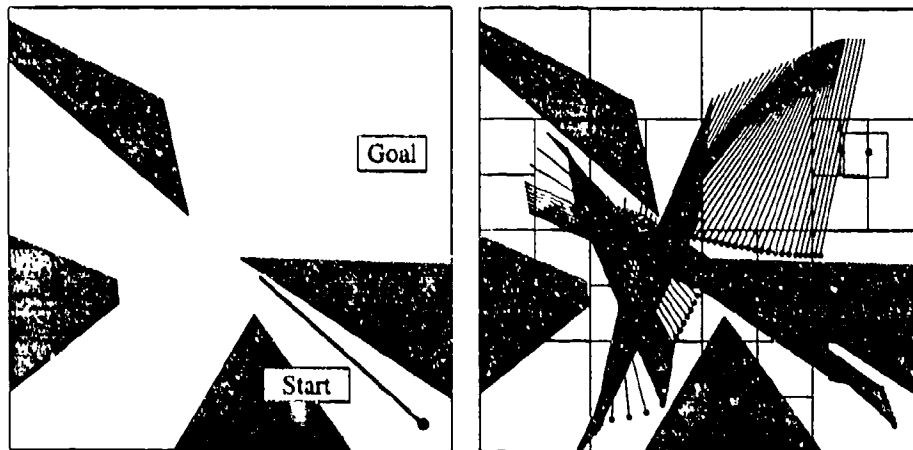
Figure 19: The partitioning after it has learned the task from 200 random start positions.

are given: the system must learn these as it explores. It takes seven trials before converging on the solution shown, which requires about two minutes run-time on a SPARC-I workstation. The exploration-length versus final-path-length ratio is 60. Interestingly, the final number of partitions is only 85. This compares very favorably with the 512 partitions which would be needed if the coarsest non-trivial uniform grid were used: $2 \times 2 \times \dots \times 2$. Unsurprisingly, for the 9-joint snake, this 512 uniform grid is too coarse, and in experiments we performed with such a grid the system became stuck, eventually deciding the problem was impossible.

5 Related work

A few other researchers in Reinforcement Learning have attempted to overcome dimensionality problems by decompositions of state space. An early attempt was [Simons *et al.*, 1982] who attempted it for 3-degree-of-freedom force control. Their method gradually learned by recording cumulative statistics of performance in partitions. More recently, we produced a variable resolution dynamic programming method [Moore, 1991]. This enabled conventional dynamic programming to be performed in real valued multivariate state spaces where straightforward discretization would fall prey to the curse of dimensionality. This is another approach to partitioning state space but has the drawback that, unlike parti-game, it requires a guess at an initially valid trajectory through state space. [Chapman and Kaelbling, 1991] proposed an interesting algorithm which used more sophisticated statistics to decide which attributes to split. Their objectives were very hard because

Figure 20: A problem with a planar rod being guided past obstacles. The state-space is three-dimensional: two values specify the position of the rod's center, and the third specifies the rod's angle from the horizontal. The angle is constrained so that the pole's dotted end must always be below the other end. The pole's center may be moved a short distance (up to $1/40$ of the diagram width) and its angle may be altered by up to 5 degrees, provided it does not hit a barrier in the process. Parti-game converged to the path shown below after two trials, with 18 times as many exploration steps as there are steps in the final path. The partitioning lines on the second diagram only show a two-dimensional slice of the full partitioning.

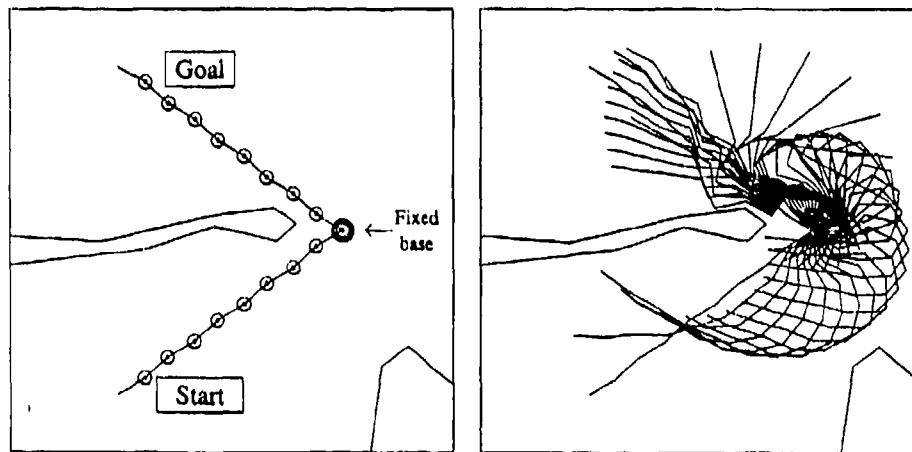


Trials	1	2	3	4	5	6	7	8	9	10
Steps	2975	189	187	no further						
Partitions	149	149	149	change						

THIS
PAGE
IS
MISSING
IN
ORIGINAL
DOCUMENT

20

Figure 22: A nine-degree-of-freedom planar robot must move from the shown start configuration to the goal. The joints are shown by small circles on the left-hand diagram which depicts two configurations of the arm: the start position and the goal position. The solution entails curling, rotating and then uncurling. It may not intersect with any of the barriers, the edge of the workspace, or itself. Convergence occurred after seven trials, with 60 times as much exploration as there are steps in the final path.



Trial	1	2	3	4	5	6	7	8	9	10
Steps	1090	430	353	330	739	200	52	no further		
Partitions	41	66	67	69	78	85	85	change		

they wished to avoid remembering transitions between cells and they did not assume continuous paths through state space, and so they obtained only limited empirical success.

In [Dayan and Hinton, 1993] a 2-dimensional hierarchical partitioning was used on a grid with 64 discrete squares, and [Kaelbling, 1993] gives another hierarchical algorithm. These references both attempt a different goal than parti-game: they try to accelerate Q-learning [Watkins, 1989] by providing it with a pre-programmed abstraction of the world. The abstraction, it is noted in both cases, may sometimes indeed lead to faster learning and can improve Q-learning if there are multiple goals in the problem. In contrast, parti-game is able to build its own abstraction using geometric reasoning and so learns more quickly (typically in fewer than ten trials and a few minutes of real time) and on significantly higher dimensional problems than have been attempted elsewhere. The price parti-game pays is that it is limited to geometric abstractions, whereas both Kaelbling's and Dayan's methods may eventually be applicable to other abstraction hierarchies.

Geometric Decompositions have also been used fairly extensively in Robot Motion Planning (e.g. [Brooks and Lozano-Perez, 1983, Kambhampati and Davis, 1986]), summarized in [Latombe, 1991]. The principal difference is that the Robot Motion Planning methods all assume that a model of the environment (typically in the form of a pre-programmed list of polygons) is supplied to the system in advance so that there is no learning or exploration capability. The experiments in [Brooks and Lozano-Perez, 1983] involve a 3-degree-of-freedom navigation problem and in [Kambhampati and Davis, 1986], a fairly difficult 2-dimensional maze.

Finally, some relation can be seen between parti-game and multigrid methods (e.g. [Hoppe, 1986]) used in numerical analysis to accelerate the convergence of solutions to partial differential equations. Multigrid methods typically increase the resolution of the grid everywhere, and like robot motion planning, do not learn: the correct system dynamics must be programmed in.

6 Discussion

6.1 Splitting

Given a partition we have decided to split, which axis should be split? Algorithm (4) states that we should split along the longest axis. This begs the question of where to split in the case of ties. The current algorithm resolves ties with a fixed ordering on axes, but we could be cleverer. In Figure 23 it is clear that a vertical split would be more useful than a horizontal split. This kind of intelligent split choice, which pays attention to the locations of outcomes, would not be difficult to incorporate.

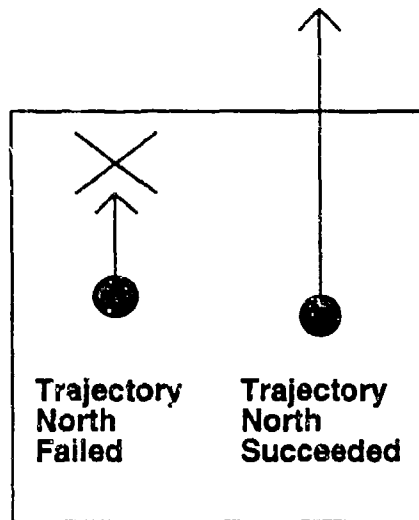


Figure 23: We have had two experiences of attempting to move North from two different points in the partition. Only one succeeded.

6.2 Not forgetting

When a partition is split, the outcomes of its children are initialized to be empty and so old data is forgotten. This is not desirable or necessary. Old trajectories could be retained and used to initialize the `OUTCOMES()` sets of those children within which earlier trajectory segments lay.

6.3 Learning the local greedy controllers

The parti-game algorithm requires that the user defines local greedy controllers. Is this not a large sacrifice of autonomy? We argue not: learning greedy controllers merely requires gathering enough local experience to form a local linear map of the low level system dynamics. This can be done with relative ease, both in a statistical and computational sense.

6.4 Dealing with an unknown goal state

There is no difficulty for parti-game in removing the assumption that the location of the goal state is known. Convergence will be considerably slowed down if it is not given, but this is not the fault of the algorithm. If there are D state variables and the goal is signaled when all state variables are simultaneously within $\pm\delta\%$ of an unknown goal value, then it is clear that an exploration of at least

$$\left(\frac{100}{2\delta}\right)^D \quad (9)$$

points on a grid in state-space are needed to ensure the goal is reached even once, whatever the learning algorithm.

A simple supplement to parti-game can be made to implement this kind of uniform exploration. It begins with a uniform grid partition with

$$\frac{100}{\delta} \quad (10)$$

breaks on each axis and encourages exploration by estimating the J_{WC} value of all unvisited partitions as zero. At this resolution at least one initial partition must be a proper subset of the goal region and so once the system has entered any part of each initial partition the goal must have been discovered.

6.5 Attaining Optimality

Parti-game is designed to find solutions to delayed reward control problems in reasonable time, without needing help in the form of initial human-supplied trajectories. The algorithm works hard to find a solution but makes no attempt to optimize it. Empirically, all solutions found have been good. There are a number of kinds of suboptimality which parti-game will not produce. In the case of navigation, for example, parti-game cannot produce loops or meanders, as shown in Figure 24.

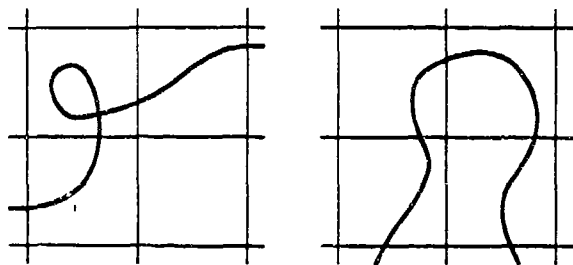


Figure 24: Partigame cannot produce either of these kinds of suboptimality. No loops, and no unblocked adjacent partitions which contain separate parts of a solution trajectory.

The lack of guaranteed optimality in parti-game is a concession to the fact that there is unlikely to be sufficient time in the lifetime of a reinforcement learning system to explore every possible solution. Future research may reveal ways to achieve weaker optimality guarantees:

- That the solution is locally optimal.
- A proof that even if the solution is not globally optimal, the global solution can be no better than factor K (in terms of cost units for the task being learned) over parti-game's solution.

Both these optimality statements will require extra assumptions about the state space. In the case of navigational problems, this can come in the form of geometric reasoning. In dynamics problems

it will be by means of local linearizations within partitions, and subsequent Linear Quadratic Gaussian (LQG) local control design (see, for example, [Sage and White, 1977]).

It is also possible that other domains will be able to use similar reasoning by means of *admissible heuristics*: a classical method in AI for formally reasoning about the optimality of proposed solutions [Nilsson, 1971].

6.6 Multiple Goals

Because it builds an explicit model of all the possible state transitions between partitions, it is a trivial matter for parti-game to change to a new goal. We have performed a number of experiments (not reported here) that confirm this.

6.7 Stochastic Dynamics

This is the hardest issue for parti-game to cope with. If a given action in a given partition produces multiple results, how do we decide if this is due to inherent randomness or due to overly coarse partitions? In the latter case it will be helpful to increase the resolution and in the former case it will not.

The easiest case will be noise in the form of

$$\text{next-state} = f(\text{state}, \text{action}) + \text{noise-signal}() \quad (11)$$

An example is an environment which randomly jogs a mobile robot between each movement. We have performed some experiments with parti-game under this scenario (not reported here), and have not yet seen it get stuck even when quite substantial noise was added. In principal, though, any amount of noise could break the partigame algorithm—if trials were run indefinitely, eventually all of state space would become partitioned to unboundedly high resolution. An improvement to parti-game might use statistical tests which try to explain outcomes in terms of location within the partition. This might help, but further research is needed.

If the randomness is something which occasionally teleports the system to a random place (breaking the assumption of paths being continuous through state space), then partigame would probably need an entirely different splitting criterion. One possibility is a version of the “G” splitting rule of [Chapman and Kaelbling, 1991].

6.8 The Curse of Dimensionality

We finish by noting a promising sign involving a series of snake robot experiments with different numbers of links (but fixed total length): Intuitively, the problem should get easier with more links.

but the curse of dimensionality would mean that (in the absence of prior knowledge) it becomes exponentially harder. This is borne out by the observation that random exploration with the three-link arm will stumble on the goal eventually, whereas the nine link robot cannot be expected to do so in tractable time. However, Figure 25 indicates that as the dimensionality rises, the amount of exploration (and hence computation) used by parti-game does not rise exponentially. It is conceivable (but not supported by further evidence in this paper) that real-world tasks may often have the same property: the complexity of the ultimate task remains roughly constant as the number of degrees of freedom increases. If so, this might be the Achilles' heel of the curse of dimensionality.

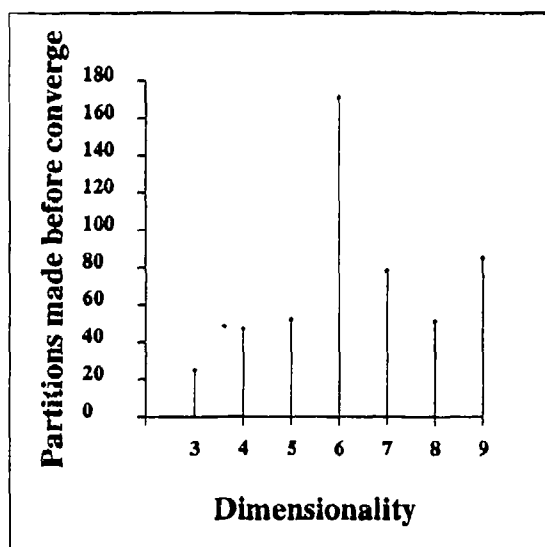


Figure 25: The number of partitions finally created against degrees of freedom for a set of snake-like robots. The partitionings built were all highly non-uniform, typically having maximum depth nodes of twice the dimensionality. The relation between exploration time and dimensionality (not shown) had a similar shape.

7 Conclusion

This paper began with the problems of coarse partitionings of state space. It then showed how worst-case assumptions can solve these problems, and very effectively identify partitions which need to have their resolutions increased. There are many interesting avenues arising from these ideas which remain open for further investigation.

References

- [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 Trans. on Systems Man and Cybernetics*,

13(5):835-846, 1983.

- [Barto *et al.*, 1991] A. G. Barto, S. J. Bradtke, and S. P. Singh. Real-time Learning and Control using Asynchronous Dynamic Programming. Technical Report 91-57, University of Massachusetts at Amherst, August 1991.
- [Bellman, 1957] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
- [Bertsekas and Tsitsiklis, 1989] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice Hall, 1989.
- [Brooks and Lozano-Perez, 1983] R. A. Brooks and T. Lozano-Perez. A Subdivision Algorithm in Configuration Space for Findpath with rotation. In *Proceedings of the 8th International Conference on Artificial Intelligence*, 1983.
- [Chapman and Kaelbling, 1991] D. Chapman and L. P. Kaelbling. Learning from Delayed Reinforcement In a Complex Domain. Technical Report, Teleos Research, 1991.
- [Dayan and Hinton, 1993] P. Dayan and G. E. Hinton. Feudal Reinforcement Learning. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, 1993.
- [Hoppe, 1986] R. H. W. Hoppe. Multi-Grid Methods for Hamilton-Jacobi-Bellman Equations. *Numerical Mathematics*, 49, 1986.
- [Kaelbling, 1993] L. Kaelbling. Hierarchical Learning in Stochastic Domains: Preliminary Results. In *Machine Learning: Proceedings of the Tenth International Workshop*. Morgan Kaufman, June 1993.
- [Kambhampati and Davis, 1986] Subbaino Kambhampati and Larry S. Davis. Multiresolution Path Planning for Mobile Robots. *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 3, 2(3), 1986.
- [Knuth, 1973] D. E. Knuth. *Sorting and Searching*. Addison Wesley, 1973.
- [Latombe, 1991] J. Latombe. *Robot Motion Planning*. Kluwer, 1991.
- [Michie and Chambers, 1968] D. Michie and R. A. Chambers. BOXES: An Experiment in Adaptive Control. In E. Dale and D. Michie, editors, *Machine Intelligence 2*. Oliver and Boyd, 1968.

- [Moore and Atkeson, 1993] A. W. Moore and C. G. Atkeson. Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time. *Machine Learning*, 13, 1993.
- [Moore, 1991] A. W. Moore. Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-valued State-spaces. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*. Morgan Kaufman, June 1991.
- [Nilsson, 1971] N. J. Nilsson. *Problem-solving Methods in Artificial Intelligence*. McGraw Hill, 1971.
- [Peng and Williams, 1993] J. Peng and R. J. Williams. Efficient Learning and Planning Within the Dyna Framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. MIT Press, 1993.
- [Sage and White, 1977] A. P. Sage and C. C. White. *Optimum Systems Control*. Prentice Hall, 1977.
- [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 Trans. on Automatic Control*, 27(5):1109-1113, October 1982.
- [Sutton, 1984] R. S. Sutton. Temporal Credit Assignment in Reinforcement Learning. Phd. thesis, University of Massachusetts, Amherst, 1984.
- [Sutton, 1990a] 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*. Morgan Kaufman, June 1990.
- [Sutton, 1990b] 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*. Morgan Kaufman, June 1990.
- [Watkins, 1989] C. J. C. H. Watkins. Learning from Delayed Rewards. Phd. Thesis, King's College, University of Cambridge, May 1989.

Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time

Andrew W. Moore

Christopher G. Atkeson

MIT Artificial Intelligence Laboratory

545 Technology Square, Cambridge, MA 02139

Abstract

We present a new algorithm, Prioritized Sweeping, for efficient prediction and control of stochastic Markov systems. Incremental learning methods such as Temporal Differencing and Q-learning have fast real time performance. Classical methods are slower, but more accurate, because they make full use of the observations. Prioritized Sweeping aims for the best of both worlds. It uses all previous experiences both to prioritize important dynamic programming sweeps and to guide the exploration of state-space. We compare Prioritized Sweeping with other reinforcement learning schemes for a number of different stochastic optimal control problems. It successfully solves large state-space real time problems with which other methods have difficulty.

1 Introduction

This paper introduces a memory-based technique, *prioritized sweeping*, which can be used both for Markov prediction and reinforcement learning. Current, model-free, learning algorithms perform well relative to real time. Classical methods such as matrix inversion and dynamic programming perform well relative to the number of observations. Prioritized sweeping seeks to achieve the best of both worlds. Its closest relation from conventional AI is the search scheduling technique of the A^* algorithm (Nilsson 1971). It is a "memory-based" method (Stanfill and Waltz 1986) in that it derives much of its power from explicitly remembering all real-world experiences. Closely related research is being performed by Peng and Williams (1992) into a similar algorithm to prioritized sweeping, which they call Dyna-Q-queue.

We begin by providing a review of the problems and techniques in Markov prediction and control. More thorough reviews may be found in Sutton (1988), Barto *et al.* (1989), Sutton (1990), Kaelbling (1990) and Barto *et al.* (1991).

A discrete, finite Markov system has S states. Time passes as a series of discrete clock ticks, and on each tick the state may change. The probability of possible successor states is a function only of the current system state. The entire system can thus be specified by S and a table of transition probabilities.

$$\begin{array}{cccc} q_{11} & q_{12} & \cdots & q_{1S} \\ q_{21} & q_{22} & \cdots & q_{2S} \\ \vdots & \vdots & & \vdots \\ q_{S1} & q_{S2} & \cdots & q_{SS} \end{array} \quad (1)$$

where q_{ij} denotes the probability that, given we are in state i , we will be in state j on the next time step. The table must satisfy $\sum_{j=1}^S q_{ij} = 1$ for every i .

Figure 1 shows an example with six states corresponding to the six cells. With the exception of the rightmost states, on each time step the system moves at random to a neighbor. For example, state 1 moves directly to state 3 with probability $\frac{1}{2}$, and thus $q_{13} = \frac{1}{2}$.

The state-space of a Markov system is partitioned into two subsets: the non-terminal states NONTERMS, and the terminal states TERMS. Once a terminal state is entered, it is never left

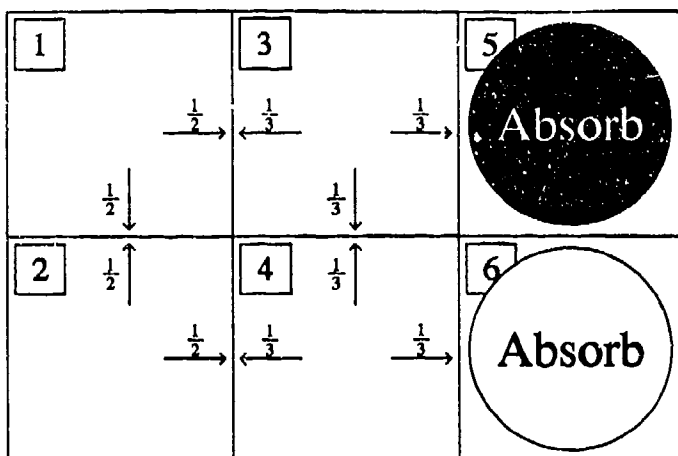


Figure 1: A six state Markov system.

($k \in \text{TERMS} \Rightarrow q_{kk} = 1$). In the example, the two rightmost states are terminal.

A Markov system is defined as *absorbing* if from every non-terminal state it is possible to eventually enter a terminal state. We restrict our attention to absorbing Markov systems.

Let us first consider questions such as “starting in state i , what is the probability of eventual absorption by terminal state k ?”. Write this value as π_{ik} . All the absorption probabilities for terminal state k can be computed by solving the following set of linear equations. Assume that the non-terminal states are indexed by $1, 2, \dots, S_{nt}$ where S_{nt} is the number of non-terminals.

$$\begin{aligned}
 \pi_{1k} &= q_{1k} + q_{11}\pi_{1k} + q_{12}\pi_{2k} + \dots + q_{1S_{nt}}\pi_{S_{nt}k} \\
 \pi_{2k} &= q_{2k} + q_{21}\pi_{1k} + q_{22}\pi_{2k} + \dots + q_{2S_{nt}}\pi_{S_{nt}k} \\
 &\vdots \\
 \pi_{S_{nt}k} &= q_{S_{nt}k} + q_{S_{nt}1}\pi_{1k} + q_{S_{nt}2}\pi_{2k} + \dots + q_{S_{nt}S_{nt}}\pi_{S_{nt}k}
 \end{aligned} \tag{2}$$

When the transition probabilities $\{q_{ij}\}$ are known it is thus an easy matter to compute the eventual absorption probabilities. Machine learning can be applied to the case in which the transition probabilities are not known in advance, and all we may do instead is watch a series of state transitions. Such a series is normally arranged into a set of trials—each trial starts in some state and then continues until the system enters a terminal state. In our example, the learner might be

shown

$$\begin{array}{l}
3 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \\
3 \rightarrow 5 \\
1 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 5 \\
\vdots
\end{array} \tag{3}$$

Learning approaches to this problem have been widely studied. A recent contribution of great relevance is an elegant algorithm called *Temporal Differencing* (Sutton 1988).

1.1 The Temporal Differencing algorithm reviewed

We describe the discrete state-space case of the temporal differencing algorithm. TD can, however, also be applied to systems with continuous state-spaces in which long term probabilities are represented by parametric function approximators such as neural networks (Tesauro 1991).

The prediction process runs in a series of epochs. Each epoch ends when a terminal state is entered. Assume we have passed through states $i_1, i_2, \dots, i_n, i_{n+1}$ so far in the current epoch. n is our age within the epoch and t is our global age. $i_n \rightarrow i_{n+1}$ is the most recently observed transition. Let $\hat{\pi}_{ik}[t]$ be the estimated value of π_{ik} after the system has been running for t state transition observations. Then the TD algorithm for discrete state-spaces updates these estimates according to the following rule:

$$\begin{array}{l}
\text{for each } i \in \text{NONTERMS (the set of non-terminal states)} \\
\quad \text{for each } k \in \text{TERMS (the set of terminal states)} \\
\quad \quad \hat{\pi}_{ik}[t+1] = \hat{\pi}_{ik}[t] + \alpha (\hat{\pi}_{i_{n+1}k}[t] - \hat{\pi}_{i_nk}[t]) \sum_{j=1}^n \lambda^{j-n} X_i(i_j)
\end{array} \tag{4}$$

where α is a learning rate parameter $0 < \alpha < 1$, where λ is a memory constant $0 \leq \lambda \leq 1$ and where

$$X_i(i_j) = \begin{cases} 1 & \text{if } i_j = i \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

In practice there is a computational trick which requires considerably less computation than the algorithm of Equation (4) but which computes the same values (Sutton 1988). The TD algorithm

then requires $O(S_t)$ computation steps per real observation, where S_t is the number of terminal states. Convergence proofs exist for several formulations of the TD algorithm (Sutton 1988; Dayan 1992).

1.2 The classical approach

The classical method proceeds by building a maximum likelihood model of the state transitions. q_{ij} is estimated by

$$\hat{q}_{ij} = \frac{\text{Number of observations } i \rightarrow j}{\text{Number of occasions in state } i} \quad (6)$$

After $t + 1$ observations the new absorption probability estimates are computed to satisfy, for each terminal state k , the $S_{nt} \times S_{nt}$ linear system

$$\hat{\pi}_{ik}[t + 1] = \hat{q}_{ik} + \sum_{j \in \text{succs}(i) \cap \text{NONTERMS}} \hat{q}_{ij} \hat{\pi}_{jk}[t + 1] \quad (7)$$

where $\text{succs}(i)$ is the set of all states which have been observed as immediate successors of i and NONTERMS is the set of non-terminal states. It is clear that if the $\hat{q}_{ik}$ estimates were correct then the solution of Equation (7) would be the solution of Equation (2).

Notice that the values $\hat{\pi}_{ik}[t + 1]$ depend only on the values of $\hat{q}_{ik}$ after $t + 1$ observations—they are not defined in terms of the previous absorption probability estimates $\hat{\pi}_{ik}[t]$. However, it is efficient to solve Equation (7) iteratively. Let $\{\rho_{ik}\}$ be a set of intermediate iteration variables containing intermediate estimates of $\hat{\pi}_{ik}[t + 1]$. What initial estimates should be used to start the iteration? An excellent answer is to use the previous absorption probability estimates $\hat{\pi}_{ik}[t]$.

The complete algorithm, performed once after every real-world observation, is shown in Figure 2. The transformation on the ρ_{ik} 's can be shown to be a *contraction mapping* as defined in Section 3.1 of Bertsekas and Tsitsiklis (1989), and thus, as the same reference proves, convergence to a solution satisfying Equation (7) is guaranteed. If, according to the estimated transitions, all states can reach a terminal state, then this solution is unique. The inner loop ("for each $k \in \text{TERMS} \dots$ ") is referred to as a probability *backup* operation, and requires $O(S_t \mu_{\text{succs}})$ basic operations, where μ_{succs} is the mean number of observed stochastic successors.

1. for each $i \in \text{NONTERMS}$, for each $k \in \text{TERMS}$,

$$\rho_{ik} := \hat{\pi}_{ik}[t]$$

2. repeat

2.1 $\Delta_{\max} := 0$

2.2 for each $i \in \text{NONTERMS}$

for each $k \in \text{TERMS}$

$$\rho_{\text{new}} = \hat{q}_{ik} + \sum_{j \in \text{succs}(i)} \hat{q}_{ij} \rho_{jk}$$

$$\Delta := |\rho_{\text{new}} - \rho_{ik}|$$

$$\rho_{ik} := \rho_{\text{new}}$$

$$\Delta_{\max} := \max(\Delta_{\max}, \Delta)$$

until $\Delta_{\max} < \epsilon$

3. for each $i \in \text{NONTERMS}$, for each $k \in \text{TERMS}$

$$\hat{\pi}_{ik}[t+1] := \rho_{ik}$$

Figure 2: Stochastic prediction with full Gauss-Seidel iteration.

Gauss-Seidel is an expensive algorithm, requiring $O(S_{nt})$ backups per real-world observation for the inner loop 2.2 alone. The absorption predictions before the most recent observation, $\hat{\pi}_{ik}[t]$, normally provide an excellent initial approximation, and only a very few iterations are required. However, when an “interesting” observation is encountered, for example a previously never-experienced transition to a terminal state, many iterations, perhaps more than S_{nt} , are needed for convergence.

2 Prioritized Sweeping

Prioritized sweeping is designed to perform the same task as Gauss-Seidel iteration while using careful bookkeeping to concentrate all computational effort on the most “interesting” parts of the system. It operates in a similar computational regime as the Dyna architecture (Sutton 1990), in which a fixed, but non-trivial, amount of computation is allowed between each real-world observation. Peng and Williams (1992) are exploring a closely related approach to prioritized sweeping, developed from Dyna and Q-learning (Watkins 1989).

Prioritized sweeping uses the Δ value from the probability update step 2.2 in the previous algorithm to determine which other updates are likely to be “interesting”—if the step produces a large change in the state’s absorption probabilities then it is interesting because it is likely that the absorption probabilities of the predecessors of the state will change given an opportunity. If, on the other hand, the step produces a small change then we will assume that there is less urgency to process the predecessors. The predecessors of a state i are all those states i' which have, at least once in the history of the system, performed a one-step transition $i' \rightarrow i$.

If we have just changed the absorption probabilities of i by Δ , then the maximum possible one-processing-step change in predecessor i' caused by our change in i is $\hat{q}_{i'i}\Delta$. This value is the priority P of the predecessor i' , and if i' is not currently on the priority queue it is placed there at priority P . If it is already on the queue, but at lower priority, then it is promoted.

After each real-world observation $i \rightarrow j$, the transition probability estimate $\hat{q}_{ij}$ is updated along with the probabilities of transition to all other previously observed successors of i . Then state i is promoted to the top of the priority queue so that its absorption probabilities are updated

immediately. Next, we continue to process further states from the top of the queue. Each state that is processed may result in the addition or promotion of its predecessors within the queue. This loop continues for a preset number of processing steps or until the queue empties.

Thus if a real world observation is interesting, all its predecessors and their earlier ancestors quickly find themselves near the top of the priority queue. On the other hand, if the real world observation is unsurprising, then the processing immediately proceeds to other, more important areas of state-space which had been under consideration on the previous time step. These other areas may be different from those in which the system currently finds itself.

Let us look at the formal algorithm in Figure 3. On entry we assume the most recent state transition was from i_{recent} . We drop the $[t]$ suffix from the $\hat{\pi}_{ik}[t]$ notation.

The decision of when we are allowed further processing, at the start of Step 2, could be implemented in many ways. In our subsequent experiments the rule is simply that a maximum of β backups are permitted per real-world observation.

There are many possible priority queue implementations, including a heap (Knuth 1973), which was used in all experiments in this paper. The cost of the algorithm is

$$O(\beta S_t(\mu_{\text{succs}} + \mu_{\text{preds}} \text{PQCOST}(S_{nt}))) \quad (8)$$

basic operations, where at most β states are processed from the priority queue and $\text{PQCOST}(N)$ is the cost of accessing a priority queue of length N . For the heap implementation this is $\log_2 N$.

States are only added to the queue if their priorities are above a tiny threshold ϵ . This is a value close to the machine floating-point precision. Stopping criteria are fraught with danger, but in this paper we discuss such dangers no further except to note that in our experiments they have caused no problem.

Prioritized sweeping is a heuristic, and in this paper no formal proof of convergence, or convergence rate, is given. We expect to be able to prove convergence using techniques from asynchronous Dynamic Programming (Bertsekas and Tsitsiklis 1989) and variants of the Temporal Differencing analysis of Dayan (1992). Later, this paper gives some empirical experiments in which convergence is relatively fast.

1. Promote state i_{recent} to top of priority queue.
2. While we are allowed further processing and priority queue not empty
 - 2.1 Remove the top state from the priority queue. Call it :

$$2.2 \Delta_{\max} = 0$$

2.3 for each $k \in \text{TERMS}$

$$\begin{aligned} \rho_{\text{new}} &= \hat{q}_{ik} + \sum_{j \in \text{succs}(i) \cap \text{NONTERMS}} \hat{q}_{ij} \hat{\pi}_{jk} \\ \Delta &:= |\rho_{\text{new}} - \hat{\pi}_{ik}| \\ \hat{\pi}_{ik} &:= \rho_{\text{new}} \\ \Delta_{\max} &:= \max(\Delta_{\max}, \Delta) \end{aligned}$$

2.4 for each $i' \in \text{preds}(i)$

$$P := \hat{q}_{i'i} \Delta_{\max}$$

If $P > \epsilon$ (a tiny threshold) and if (i' is not on queue or P exceeds the current priority of i') then promote i' to new priority P .

Figure 3: The prioritized sweeping algorithm.

The memory requirements of learning the $S \times S$ transition probability matrix, where S is the number of states, may initially appear prohibitive, especially since we intend to operate with more than 10,000 states. However, we need only allocate memory for the experiences the system actually has, and for a wide class of physical systems there is not enough time in the lifetime of the system to run out of memory.

Similarly, the average number of successors and predecessors of states in the estimated transition matrix can be assumed $\ll S$. A simple justification is that few real problems are fully connected, but a deeper reason is that for large S , even if the true transition probability matrix is not sparse, there will never be time to gain enough experience for the estimated transition matrix to not be sparse.

3 A Markov Prediction Experiment

Consider the 500 state Markov system depicted in Figure 1, which is a more complex version of the problem presented in Figure 1. Appendix A gives details of how this problem was randomly generated. The system has sixteen terminal states, depicted by white and black circles. The prediction problem is to estimate, for every non-terminal state, the long term probability that it will terminate in a black, rather than a white, circle. The data available to the learner is a sequence of observed state transitions.

Temporal differencing, the classical method, and prioritized sweeping were all applied to this problem. Each learner was shown the same sequence. TD used parameters $\lambda = 0.25$ and $\alpha = 0.05$ which gave the best performance of a number of manually optimized experiments. The classical method was required to compute up to date absorption probability estimates after every real world observation. Prioritized sweeping was allowed five backups per real experience (5000) so it thus updated the v_{ij} estimates for the five highest priority states between each real world observation. The threshold for getting time changes ϵ was 10^{-5} . Each method was evaluated at a number of stages of learning by stopping the real time runs and computing the error between the estimated white absorption probabilities which we denote by $\hat{w}_{ij}$ and the true values which we denote by

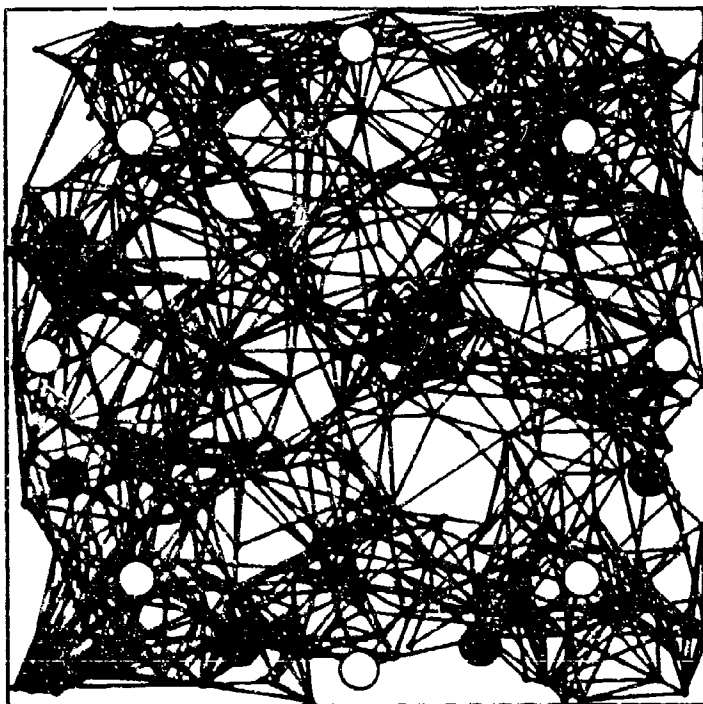


Figure 4 A 500 state Markov system. Each state has, on average, 5 stochastic successors

$\pi_{i, \text{WHITE}}$ The following RMS error over all states was recorded:

$$\sqrt{\frac{1}{S_{\text{tot}}} \sum_{i=0}^{S_{\text{tot}}} (\pi_{i, \text{WHITE}} - \hat{\pi}_{i, \text{WHITE}})^2} \quad (9)$$

In Figure 5 we look at the RMS error plotted against the number of observations. After 100,000 experiences all methods are performing well; TD is the weakest but even it manages an RMS error of only 0.1.

In Figure 6 we look at a different measure of performance: plotted against real time. Here we see the great weakness of the classical technique. Performing the Gauss-Seidel algorithm of Figure 2 after each observation gives excellent predictions but is very time consuming, and after 300 seconds there has only been time to process a few thousand observations. After the same amount of time, TD has had time to process almost half a million observations. Prioritized sweeping performs best relative to real time. It takes approximately ten times as long as TD to process each observation but because the data is used more effectively, convergence is superior.

Ten further experiments, each with a different random 500 state problem, were run. These

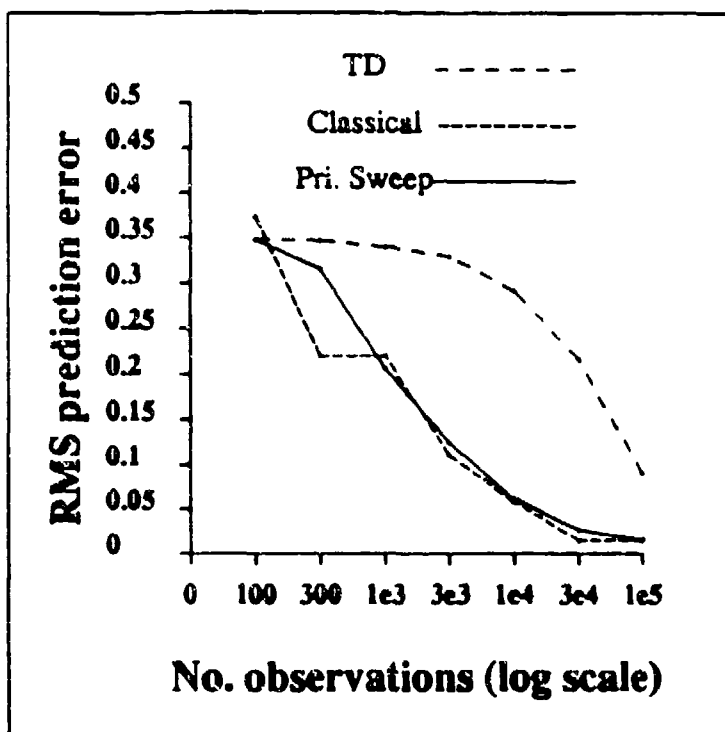


Figure 5: RMS prediction error between true absorption probabilities and predicted values, plotted against number of data-points observed. For prioritized sweeping, $\beta = 5$ and $\epsilon = 10^{-5}$.

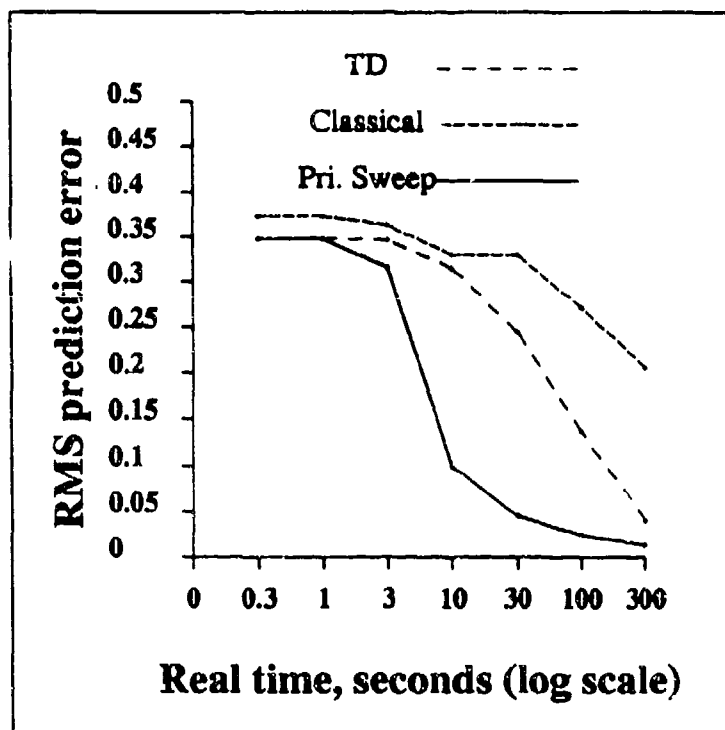


Figure 6: RMS prediction error between true absorption probabilities and predicted values, plotted against real time, in seconds, running the problem on a Sun-4 workstation.

	TD	Classical	Pri. Sweep
After 100,000 observations	0.14 ± 0.077	0.024 ± 0.0063	0.024 ± 0.0061
After 300 seconds	0.079 ± 0.067	0.23 ± 0.038	0.021 ± 0.0080

Table 1: RMS prediction error: mean and standard deviation for ten experiments.

further runs, the final results of which are given in Table 1, indicate that the graphs in Figures 5 and 6 are not atypical.

This example has shown the general theme of this paper. Model-free methods perform well in real time but make weak use of their data. Classical methods make good use of their data but are often impractically slow. Techniques such as prioritized sweeping are interesting because they may be able to achieve both.

There is an important footnote concerning the classical method. If the problem had only required that a prediction be made after all transitions had been observed, then the only real time cost would have been recording the transitions in memory. The absorption probabilities could then have been computed as an individual large computation at the end of the sequence, giving the best possible estimate with a relatively small overall time cost. For the 500-state problem, we estimate the cost as approximately 30 seconds for 100,000 points. Prioritized sweeping could also benefit from only being required to predict after seeing all the data, although with little advantage over the simpler, classical algorithm. Prioritized sweeping is thus more usefully applicable to the class of tasks in which a prediction is required on every time step. Furthermore, the remainder of the paper concerns control of Markov decision tasks, in which the maintenance of up to date predictions is particularly beneficial.

4 Learning Control of Markov Decision Tasks

Let us consider a related stochastic prediction problem, which bridges the gap between Markov prediction and control. Suppose the system gets rewarded for entering certain states and punished for entering others. Let the reward of the i th state be r_i . An important quantity is then the

expected discounted reward-to-go of each state. This is an infinite sum of expected future rewards, with each term supplemented by an exponentially decreasing weighting factor γ^k where γ is called the *discount factor*. The expected discounted reward-to-go is

$$J_i = \begin{aligned} & (\text{This reward}) + \\ & \gamma \quad (\text{Expected reward in 1 time step}) + \\ & \gamma^2 \quad (\text{Expected reward in 2 time steps}) + \\ & \vdots \\ & \gamma^k \quad (\text{Expected reward in } k \text{ time steps}) + \\ & \vdots \end{aligned} \quad (10)$$

For each i , J_i can be computed recursively as a function of its immediate successors.

$$\begin{aligned} J_1 &= r_1 + \gamma (q_{11}J_1 + q_{12}J_2 + \dots + q_{1S}J_S) \\ J_2 &= r_2 + \gamma (q_{21}J_1 + q_{22}J_2 + \dots + q_{2S}J_S) \\ &\vdots \\ J_S &= r_S + \gamma (q_{S1}J_1 + q_{S2}J_2 + \dots + q_{SS}J_S) \end{aligned} \quad (11)$$

which is another set of linear equations that may be solved if the transition probabilities q_{ij} are known. If they are not known, but instead a sequence of state transitions and r_i observations is given, then slight modifications of TD, the classical algorithm, and prioritized sweeping can all be used to estimate J_i .

Markov decision tasks

Markov decision tasks are an extension of the Markov model in which, instead of passively watching the state move around randomly, we are able to influence it.

Associated with each state, i , is a finite, discrete set of actions, $\text{actions}(i)$. On each time step, the controller must choose an action. The probabilities of potential next states depend not only on the current state, but also on the chosen action. We will supplement our example problem with actions:

$$\begin{aligned} \text{actions}(1) &= \{\text{RANDOM}, \text{RIGHT}\} & \text{actions}(3) &= \{\text{RANDOM}, \text{RIGHT}\} & \text{actions}(5) &= \{\text{STAY}\} \\ \text{actions}(2) &= \{\text{RANDOM}, \text{RIGHT}\} & \text{actions}(4) &= \{\text{RANDOM}, \text{RIGHT}\} & \text{actions}(6) &= \{\text{STAY}\} \end{aligned} \quad (12)$$

where RANDOM causes the same random transitions as before, RIGHT moves, with probability 1, to the cell immediately to the right, and STAY makes us remain in the same state. There is still no escape from states 5 and 6.

We use the notation q_{ij}^a for the probability that we move to state j , given that we have commenced in state i and applied action a . Thus, in our example $q_{13}^{\text{RANDOM}} = \frac{1}{2}$ and $q_{13}^{\text{RIGHT}} = 1$.

A *policy* is a mapping from states to actions. For example, Figure 7 shows the policy

$$\begin{aligned} 1 \rightarrow \text{RIGHT} \quad 3 \rightarrow \text{RIGHT} \quad 5 \rightarrow \text{STAY} \\ 2 \rightarrow \text{RANDOM} \quad 4 \rightarrow \text{RANDOM} \quad 6 \rightarrow \text{STAY} \end{aligned} \quad (13)$$

If the controller chooses actions according to a fixed policy then it behaves like a Markov system. The expected discounted reward-to-go can then be defined and computed in the same manner as Equation (11).

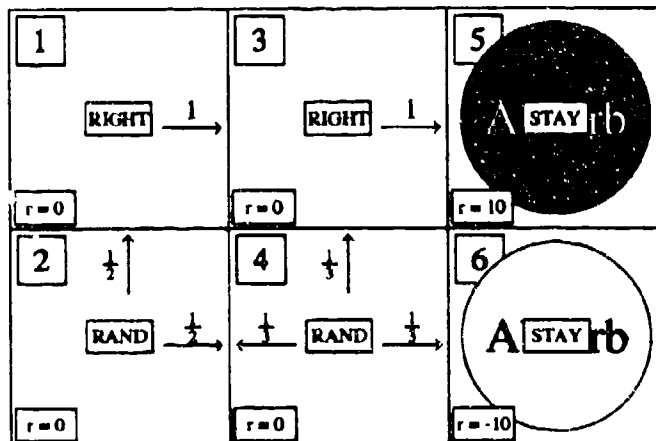


Figure 7: The policy defined by Equation (13). Also shown is a reward function (bottom left of each cell). Large expected reward-to-go involves getting to '5' and avoiding '6'.

If the goal is large reward-to-go, then some policies are better than others. An important result from the theory of Markov decision tasks tells us that there always exists at least one policy which is *optimal* in the following sense. For every state, the expected discounted reward-to-go using an optimal policy is no worse than that from any other policy.

Furthermore, there is a simple algorithm for computing both an optimal policy and the expected discounted reward-to-go of this policy. The algorithm is called *Dynamic Programming* (Bellman 1957). It is based on the following relationship known as *Bellman's optimality equation* which holds be-

tween the optimal expected discounted reward-to-go at different states.

$$J_i = \max_{a \in \text{actions}(i)} (r_i + \gamma(q_{i1}^a J_1 + q_{i2}^a J_2 + \cdots + q_{iS}^a J_S)) \quad (14)$$

Dynamic programming applied to our example gives the policy shown in Figure 7, which happens to be the unique optimal policy.

A very important question for machine learning has been how to obtain an optimal, or near optimal, policy when the q_{ij}^a values are not known in advance. Instead, a series of actions, state transitions, and rewards is observed. For example:

$$\begin{aligned} 1(r_1 = 0) &\xrightarrow{\text{RANDOM}} 2(r_2 = 0) \xrightarrow{\text{RANDOM}} 4(r_4 = 0) \xrightarrow{\text{RIGHT}} 6(r_6 = 10) \\ 2(r_2 = 0) &\xrightarrow{\text{RANDOM}} 1(r_1 = 0) \xrightarrow{\text{RIGHT}} 3(r_3 = 0) \xrightarrow{\text{RANDOM}} 5(r_5 = -10) \\ 3(r_3 = 0) &\xrightarrow{\text{RANDOM}} 5(r_5 = 10) \\ &\vdots \end{aligned} \quad (15)$$

A critical difference between this problem and the Markov prediction problem of the earlier sections is that the controller now affects which transitions are seen, because it supplies the actions.

The question of learning such systems is studied by the field of *reinforcement learning*, which is also known as “learning control of Markov decision tasks”. Early contributions to this field were the checkers player of Samuel (1959) and the BOXES system of Michie and Chambers (1968). Even systems which may at first appear trivially small, such as the two armed bandit problem (Berry and Fristedt 1985) have promoted rich and interesting work in the statistics community.

The technique of gradient descent optimization of neural networks in combination with approximations to the policy and reward-to-go (called the “adaptive heuristic critic”) was introduced by Sutton (1984). Kaelbling (1990) introduced several applicable techniques, including the *Interval Estimation algorithm*. Watkins (1989) introduced an important model-free asynchronous Dynamic Programming technique called Q-learning. Sutton (1990) has extended this further with the Dyna architecture. Christiansen *et al.* (1990) applied a planner, closely related to Dynamic Programming, to a tray tilting robot. An excellent review of the entire field may be found in (Barto *et al.* 1991).

4.1 Prioritized sweeping for learning control of Markov decision tasks

The main differences between this case and the previous application of prioritized sweeping are

1. We need to estimate the optimal discounted reward-to-go, J , of each state, rather than the eventual absorption probabilities.
2. Instead of using the absorption probability backup Equation (6), we use Bellman's equation (Bellman 1957; Bertsekas and Tsitsiklis 1989):

$$\hat{J}_i = \max_{a \in \text{actions}(i)} \left(\hat{r}_i^a + \gamma \times \sum_{j \in \text{succs}(i,a)} \hat{q}_{ij}^a \hat{J}_j \right) \quad (16)$$

where $\hat{J}_i$ is the estimate of the optimal discounted reward starting from state i , γ is the discount factor, $\text{actions}(i)$ is the set of possible actions in state i , and $\hat{q}_{ij}^a$ is the maximum likelihood estimated probability of moving from state i to state j given that we have applied action a . The estimated immediate reward, $\hat{r}_i^a$, is computed as the mean reward experienced to date during all previous applications of action a in state i .

3. The rate of learning can be affected considerably by the controller's exploration strategy.

The algorithm for prioritized sweeping in conjunction with Bellman's equation is given in Figure 8. The only substantial difference between this algorithm and the prediction case is the state backup step, namely the Bellman's equation application of Step 2.2. Notice also that the predecessors of a state are now a set of state-action pairs.

Let us now consider the question of how best to gain useful experience in a Markov decision task. The formally correct method would be to compute that exploration which maximizes the expected reward received over the robot's remaining life. This computation, which requires a prior probability distribution over the space of Markov decision tasks, is unrealistically expensive. It is computationally exponential in all of (i) the number of time steps for which the system is to remain alive (ii) the number of states in the system, and (iii) the number of actions available (Berry and Fristedt 1985).

An exploration heuristic is thus required. Kaelbling (1990) and Barto *et al.* (1991) both give excellent overviews of the wide range of heuristics which have been proposed.

1. Promote state i_{recent} to top of priority queue.
2. While we are allowed further processing and priority queue not empty
 - 2.1 Remove the top state from the priority queue. Call it i
 - 2.2 $\rho_{\text{new}} := \max_{a \in \text{actions}(i)} \left(\hat{r}_i^a + \gamma \times \sum_{j \in \text{succs}(i,a)} \hat{q}_{ij}^a \hat{J}_j \right)$
 - 2.3 $\Delta_{\text{max}} := |\rho_{\text{new}} - \hat{J}_i|$
 - 2.4 $\hat{J}_i := \rho_{\text{new}}$
 - 2.5 for each $(i', a') \in \text{preds}(i)$

$$P := \hat{q}_{i'i}^{a'} \Delta_{\text{max}}$$

If $P > \epsilon$ (a tiny threshold) and if (if i' not on queue or P exceeds the current priority of i') then promote i' to new priority P .

Figure 8: The prioritized sweeping algorithm for Markov Decision Tasks.

We use the philosophy of *optimism in the face of uncertainty*, a method successfully developed by the Interval Estimation (IE) algorithm of Kaelbling (1990) and by the exploration bonus technique in Dyna (Sutton 1990). The same philosophy is also used by Thrun and Möller (1992).

A slightly different heuristic is used with the prioritized sweeping algorithm. This is because of minor problems of computational expense for IE and the instability of the exploration bonus in large state-spaces.

The slightly different optimistic heuristic is as follows. In the absence of contrary evidence, any action in any state is assumed to lead us directly to a fictional absorbing state of permanent large reward r^{opt} . The amount of evidence to the contrary which is needed to quench our optimism is a system parameter, T_{bored} . If the number of occurrences of a given state-action pair is less than T_{bored} , we assume that we will jump to fictional state with subsequent long term reward $r^{\text{opt}} + \gamma r^{\text{opt}} + \gamma^2 r^{\text{opt}} + \dots = r^{\text{opt}}/(1 - \gamma)$. If the number of occurrences is not less than T_{bored} , then we use the true, non-optimistic, assumption. Thus the optimistic reward-to-go estimate $\hat{J}^{\text{opt}}$ is

$$\hat{J}_i^{\text{opt}} = \max_{a \in \text{actions}(i)} \begin{cases} r^{\text{opt}}/(1 - \gamma) & \text{if } n_i^a < T_{\text{bored}} \\ \hat{r}_i^a + \gamma \times \sum_{j \in \text{succs}(i,a)} \hat{q}_{ij}^a \hat{J}_j^{\text{opt}} & \text{otherwise} \end{cases} \quad (17)$$

where n_i^a is the number of times action a has been tried to date in state i . The important feature, identified by Sutton (1990), is the *planning to explore* behavior caused by the appearance of the optimism on both sides of the equation. A related exploration technique was used by (Christiansen *et al.* 1990). Consider the situation in Figure 9. The top left hand corner of state-space only looks attractive if we use an optimistic heuristic. The areas near the frontiers of little experience will have high $\hat{J}^{\text{opt}}$, and in turn the areas near those have nearly as high $\hat{J}^{\text{opt}}$. Therefore, if prioritized sweeping (or any other asynchronous dynamic programming method) does its job, from START we will be encouraged to go north towards the unknown instead of east to the best reward discovered to date.

The system parameter r^{opt} does not require fine tuning. It can be set to a gross overestimate of the largest possible reward, and the system will simply continue exploration until it has sampled all state-action combinations T_{bored} times. However, Section 6 discusses its use as a search-guiding heuristic similar to the heuristic at the heart of A^* search.

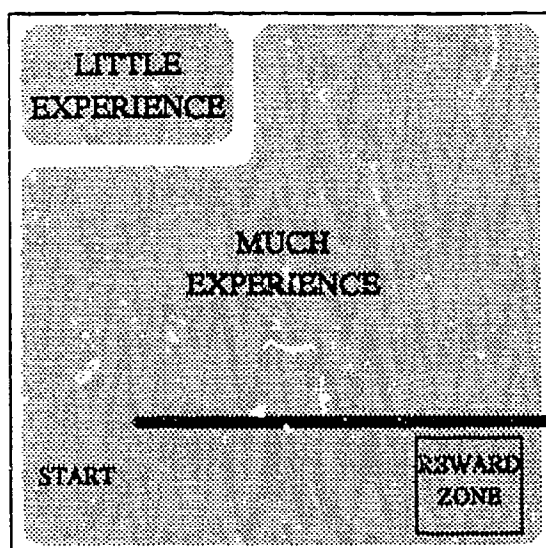


Figure 9: The state-space of a very simple path planning problem.

The T_{bored} parameter, which defines how often we must try a given state-action combination before we cease our optimism, certainly does require forethought by the human programmer. If too small, we might overlook some low probability but highly rewarding stochastic successor. If too high, the system will waste time needlessly resampling already reliable statistics. Thus, the exploration procedure does not have full autonomy. This is, arguably, a necessary weakness of any non-random exploration heuristic. Dyna's exploration bonus contains a similar parameter in the relative size of the exploration bonus to the expected reward, and Interval Estimation has the parameter implicit in the optimistic confidence level.

The selection of an appropriate T_{bored} would be hard to formalize. It should take into account: the expected lifetime of the system, a measure of the importance of not becoming stuck during learning, and perhaps any available prior knowledge of the stochasticity of the system, or known constraints on the reward function. An automatic procedure for computing T_{bored} would require a formal definition of the human programmer's requirements and a prior distribution of possible worlds.

5 Experimental Results

This section begins with some comparative results in the familiar domain of stochastic two dimensional maze worlds. It then examines the β parameter which specifies the amount of computation (number of Bellman equation backups) allowed per real-world observation and also the T_{bored} parameter which defines how much exploration is performed. A number of larger examples are then used to investigate performance for a range of different discrete stochastic reinforcement tasks.

Maze problems

Each state has four actions: one for each direction. Blocked actions do not move. One goal state (the star in subsequent figures) gives 100 units of reward, all others give no reward, and there is a discount factor of 0.99. Trials start in the bottom left corner. The system is reset to the start state whenever the goal state has been visited ten times since the last reset. The reset is outside the learning task: it is not observed as a state transition.

Dyna and prioritized sweeping were both allowed ten Bellman's equation backups per observation ($\beta = 10$). Two versions of Dyna were tested:

1. Dyna-PI+ is the original Dyna-PI of Sutton (1990), supplemented with the exploration bonus ($\epsilon = 0.001$) from the same paper.
2. Dyna-opt is the original Dyna-PI supplemented with the same T_{bored} optimistic heuristic that is used by prioritized sweeping.

Table 2 shows the number of observations before convergence. A trial was defined to have converged by a given time if no subsequent sequence of 1000 decisions contained more than 2% suboptimal decisions. The test for optimality was performed by comparison with the control law obtained from full dynamic programming using the true simulation.

We begin with some results for deterministic problems, in the first three rows of Table 2. The first row shows that Dyna-PI+ converged for all problems except the 4,528 state problem. A smaller exploration bonus than $\epsilon = 0.001$ might have helped the latter problem converge, albeit slowly.

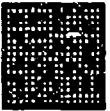
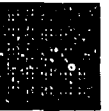
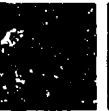
Maze							
	15 state	117 state	178 state	284 state	605 state	2627	4528
Det Dyna-PI+	400	500	10,000	18,000	36,000	195,000	$> 10^6$
Det Dyna-opt	300	900	4,250	12,000	21,000	105,000	245,000
Det PriSweep	150	1,200	3,250	2,800	6,000	29,000	59,000
Stc Q	600	31,000	62,000	310,000	untested	untested	untested
Stc Q-opt	500	$> 10^6$	$> 10^6$	untested	untested	untested	untested
Stc Dyna-PI+	400	4750	12,000	25,000	58,000	240,000	525,000
Stc Dyna-opt	700	5250	7500	14,000	35,000	155,000	310,000
Stc PriSweep	600	3500	5500	11,000	22,000	94,000	200,000

Table 2. Number of observations before 98% of decisions were subsequently optimal. These values have been rounded. For prioritized sweeping (and Dyna, where applicable) $\beta = 10$, $\epsilon = 10^{-3}$ and $\tau^{\text{opt}} = 200$. The tabulated experiments were all only run once; however, further multiple runs of the optimistic Dyna and prioritized sweeping have revealed little variance in convergence rate. See also Figures 11 and 14.

The other two rows used the optimistic heuristic with $r^{opt} = 200$ and $T_{bored} = 1$. The r^{opt} value thus overestimated the best possible reward by a factor of two—this was to see if we would converge without an accurate estimation of the true best possible reward. $T_{bored} = 1$ meant that as soon as something was tried all optimism was lost. This is a safe strategy in a deterministic environment.

The learning controller was given no clues beyond those implicit in the two parameters r^{opt} and T_{bored} . Thus, to ensure convergence to the optimum, it *had* to sample each state-action pair at least once.

Prioritized sweeping required fewer steps than optimistic Dyna in all mazes but one small one. All learners and runs took between 10–30 seconds per thousand observations running on a Sun-4 workstation. Interestingly, prioritized sweeping usually took about half the real time of Dyna. This is because during much of the exploration there were so few surprises that it did not need to use its full allocation of Bellman's equation processing steps. This effect is even more pronounced if 300 processing steps per observation are allowed instead of ten. For example, in the 4,528 state problem, optimistic Dyna then required 143,000 observations and took three hours. Prioritized sweeping required 21,000 observations and took fifteen minutes.

The lower part of Table 2 shows the results for stochastic problems using the same mazes. Each action had a 50% chance of being corrupted to a random value before it was applied. Thus if "North" was applied the outcome was movement North $\frac{1}{2} + \frac{1}{8} = \frac{5}{8}$ of the time, and each other direction $\frac{1}{8}$ of the time. Prioritized sweeping and optimistic Dyna each used a T_{bored} value of 5. Thus, they sampled every state-action combination five times before losing their optimism. This value was chosen as a reasonable balance between exploration and exploitation, given the authors' knowledge of the stochasticity of the system, and happily it proved to be satisfactory. As we discussed in Section 4.1, the choice of T_{bored} is not automated for any of these experiments.

These stochastic results also include a recent interesting incremental technique called Q-learning (Watkins 1989), which manages to learn without constructing a state transition model. Additionally, we tried Q-learning using the same T_{bored} optimistic heuristic as prioritized sweeping. The initial Q values were set high to encourage better initial exploration than a random walk. Much effort was put into tuning Q for this application. Its performance was, however, worse. In particu-

lar, the optimistic heuristic is a disaster for Q learning which easily gets trapped—this is because Q learning only pays attention to the current state of the system while the “planning to explore” behavior requires that attention is paid to areas of the state-space which the system is not currently in.

For the stochastic maze results the difference between optimistic Dyna and prioritized sweeping is less pronounced. This is because the large number of predecessors quickly dilute the wave of interesting changes which are propagated back on the priority queue, leading to a queue of many, very similar, priorities. However, prioritized sweeping still required less than half the total real time of either version of Dyna before convergence.

A small, fully connected, example

We also have results for a five state bench-mark problem described by Sato *et al.* (1988) and also used in Barto and Singh (1990). The transition matrix is in Figure 10 and the results are shown in Table 3. A T_{bored} parameter of 20 was used. In fact, $T_{\text{bored}} = 5$ also converged 20 times out of 20, taking on average 120 steps and therefore $T_{\text{bored}} = 20$ was considered a safe safety margin. The two Q-learners were heavily tweaked to find their best performance. The EQ-algorithm (Barto and Singh 1990) is designed to guarantee convergence at all costs—and so its poor comparative performance here is to be expected. Dyna-PI+ was given what was probably too small an exploration bonus for the problem. The reduced exploration meant faster convergence, but on one occasion some misleading early transitions caused it to get stuck with a suboptimal policy.

The system parameters for prioritized sweeping

We now look at two results to give insight into two important parameters of prioritized sweeping. Firstly we consider its performance relative to the number of backups per observation. This experiment used the stochastic, 605 state example from Table 2 and the results are graphed in Figure 11. Using one operation is almost equivalent to optimistic Q-learning which does not converge. Even using only two backups gives reasonable performance, and performance improves as the number of

Q	Q-opt	Dyna-PI+	Dyna-opt	PriSweep	EQ
$2,105 \pm 520$	$2,078 \pm 430$	252 ± 35 (one failure)	470 ± 21	472 ± 22	$7,105 \pm 662$

Table 3. The mean number of observations before $> 98\%$ of subsequent decisions were optimal. Each learner was run twenty times and in all cases, bar one, there was eventual convergence to optimal performance. Also shown is the standard deviation of the twenty trials. The discount factor was $\gamma = 0.8$. For the optimistic methods $r^{\text{opt}} = 10$ and $T_{\text{bored}} = 20$. For prioritized sweeping and Dyna $\beta = 10$, and for prioritized sweeping $\epsilon = 10^{-3}$.

	R	a=0	a=1	a=2	a=3	a=4
a=0,0	-0.4	10	30	20	20	30
a=0,1	0	20	20	20	20	20
a=0,2	1.2	10	10	10	30	40
a=1,0	-2.5	10	10	20	30	30
a=1,1	0.2	10	10	40	10	10
a=1,2	-1.3	10	30	20	10	10
a=2,0	1.3	10	40	20	20	10
a=2,1	0.8	30	20	20	20	10
a=2,2	-1.4	30	20	10	10	30
a=3,0	-1.2	20	30	10	10	10
a=3,1	-0.7	30	10	10	40	10
a=3,2	0.1	20	30	30	10	10
a=4,0	2.4	20	20	20	30	20
a=4,1	-1.7	20	30	20	10	20
a=4,2	1	10	40	20	10	20

Figure 10: Transition probabilities ($\times 100$) and expected rewards of a five state, three action, Markov control problem.

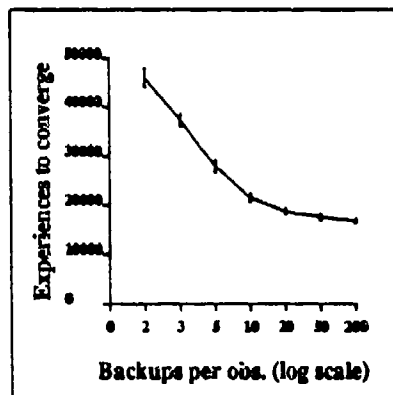


Figure 11: Number of experiences needed for prioritized sweeping to converge, plotted against number of backups per observation (β). This used the 605 state stochastic maze from Table 2 ($\gamma = 0.99$, $r^{\text{opt}} = 200$, $T_{\text{bored}} = 5$, $\epsilon = 10^{-3}$). The error bars show the standard deviations from ten runs with different random seeds.

backups increases. Beyond fifty backups, the priority queue usually gets exhausted on each time step, and there is little further improvement.

The other parameter is T_{bored} . We use a test case in which inadequate exploration is particularly

dangerous. The maze in Figure 12 has two reward states. The lesser reward of 50 comes from the state in the bottom right. The greater reward of 100 is from the more inaccessible state near the top right. Trials always begin from the bottom left and the world is stochastic in the same manner as the earlier examples. Trials are reset when either goal state is encountered ten times. If T_{bored} is set too low and if there is bad luck while attempting to explore near the large reward state then the controller will lose interest, never return, and very likely spend the rest of its days traveling to the inferior reward. Each value of T_{bored} was run ten times and we recorded the percentage of runs which had converged correctly by 50,000 observations. Figure 13 graphs the results. For this problem $T_{\text{bored}} = 5$ (which was checked a further 30 times) appears sufficient to ensure that we do not become stuck.

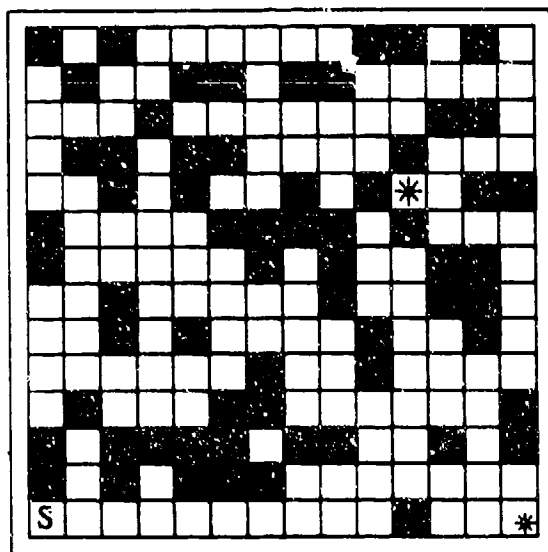


Figure 12: A misleading maze.
A small reward in the bottom,
right tempts us away from a
larger reward.

Figure 14 shows the number of experiences needed for convergence as a function of T_{bored} for the same set of experiments.

Other tasks

We begin with a task with a 3-d state-space quantized into 14,400 potential discrete states: guiding a rod through a planar maze by translation and rotation. There are four actions: move forwards one unit along the rod's length, move backwards one unit, rotate left one unit and rotate right one

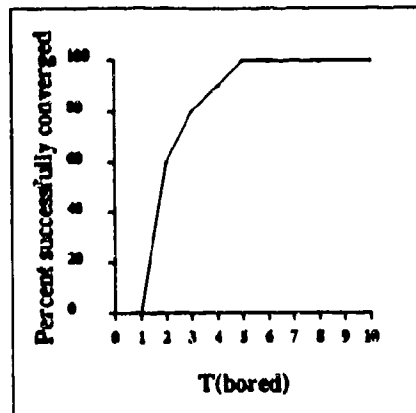


Figure 13: The frequency of correct convergence versus T_{bored} for the misleading maze ($\gamma = 0.99$, $r^{\text{opt}} = 200$, $\beta = 10$, $\epsilon = 10^{-3}$).

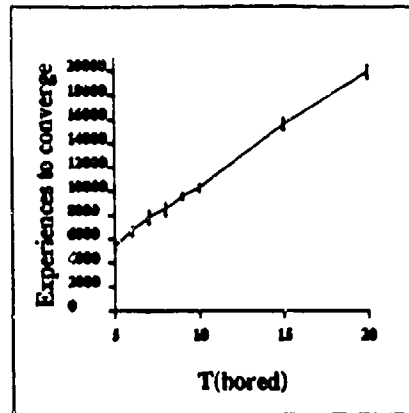


Figure 14: The mean and standard deviation number of experiences before convergence for ten independent experiments, as a function of T_{bored} for the misleading maze. Parameter values : as in Figure 13.

unit. In fact, the action takes us to the nearest quantized state after having applied the action. There are 20×20 position quantizations and 36 angle quantizations producing 14,400 states, though many are unreachable from the start. The distance unit is $1/20$ th the width of the workspace and the angular unit is 10 degrees. The problem is deterministic but requires a long, very specific, sequence of moves to get to the goal. Figure 15 shows the problem, obstacles and shortest solution for our experiments.

Q, Dyna-PI+, Optimistic Dyna and prioritized sweeping were all tested. The results are in Table 4.

Q and Dyna-PI+ did not even travel a quarter of the way to the goal, let alone discover an optimal path, within 200,000 experiences. It is possible that a very well-chosen exploration bonus would have helped Dyna-PI+ but in the four different experiments we tried, no value produced stable exploration.

Optimistic Dyna and prioritized sweeping both eventually converged, with the latter requiring a third the experiences and a fifth the real time.

When 2000 backups per experience were permitted, instead of 100, then both optimistic Dyna

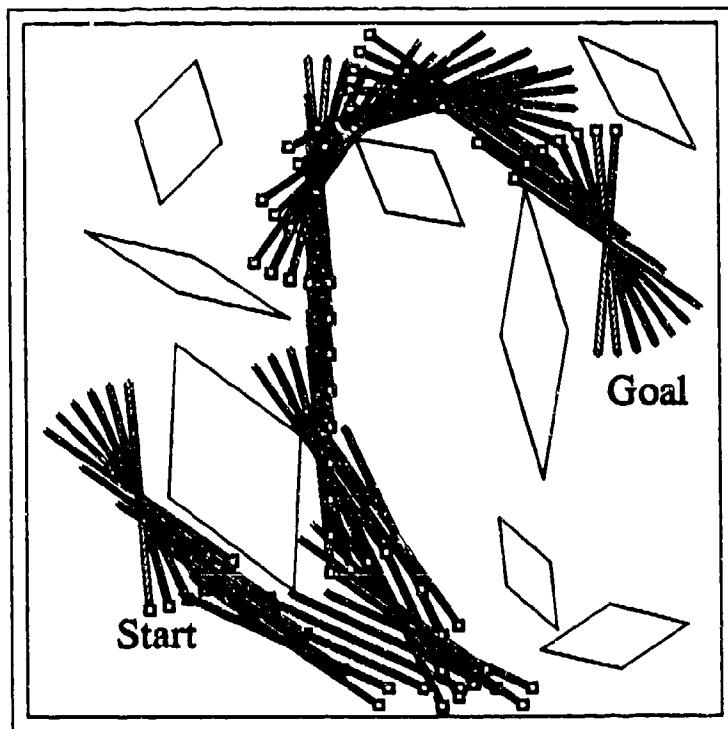


Figure 15: A three-DOF problem and the shortest solution path.

	Experiences to converge	Real time to converge
Q	never	
Dyna-FI+	never	
Optimistic Dyna	55,000	1500 secs
Prioritized Sweeping	14,000	330 secs

Table 4. Performance on the deterministic rod-in-maze task. Both Dynas and prioritized sweeping were allowed 100 backups per experience ($\gamma = 0.99, r^{\text{opt}} = 200, \beta = 100, T_{\text{bored}} = 1, \epsilon = 10^{-3}$).

and prioritized sweeping required fewer experiences to converge. Optimistic Dyna took 21,000 experiences instead of 55,000 but took 2,900 seconds—almost twice the real time. Prioritized sweeping took 13,500 instead of 14,000 experiences—very little improvement, but it used no extra time. This indicates that for prioritized sweeping, 100 backups per observation is sufficient to make almost complete use of its observations, so that all the long term reward ($\hat{J}_i$) estimates are very close to the estimates which would be globally consistent with the transition probability estimates ($\hat{q}_{ij}^a$). Thus, we conjecture that even full dynamic programming after each experience (which would take days of real time) would do little better.

We also consider a more complex extension of the maze world, invented by Singh (1991), which consists of a maze and extra state information dependent on where you have visited so far in the maze. We use the example in Figure 16. There are 263 cells, but there are also four binary flags appended to the state, producing a total of $263 \times 16 = 4208$ states. The flags, named A, B, C and X, are set whenever the cell containing the corresponding letter is passed through. All flags are cleared when the start state (in the bottom left hand corner) is entered. A reward is given when the goal state (top right) is entered, only if flags A, B and C are set. Flag X provides further interest. If X is clear, the reward is 100 units. If X is set, the reward is only 50 units. This task does not specify which order A, B and C are to be visited. The controller must find the optimal path.

Prioritized sweeping was tried with both the deterministic and stochastic maze dynamics ($\gamma = 0.99, \tau^{\text{opt}} = 200, \beta = 10, \epsilon = 10^{-3}$). In the deterministic case $T_{\text{bored}} = 1$. In the stochastic case $T_{\text{bored}} = 5$. In both cases it found the globally optimal path through the three good flags to the goal, avoiding flag X. The deterministic case took 19,000 observations and twenty minutes of real time. The stochastic case required 120,000 observations and two hours of real time.

In these experiments, no information regarding the special structure of the problem was available to the learner. For example, knowledge of the cell at coordinates (7, 1) with flag A set had no bearing on knowledge of the cell at coordinates (7, 1) with A clear. If we told the learner that cell transitions are independent of flag settings then the convergence rate would be increased considerably. A far more interesting possibility is the automatic discovery of such structure by inductive inference on

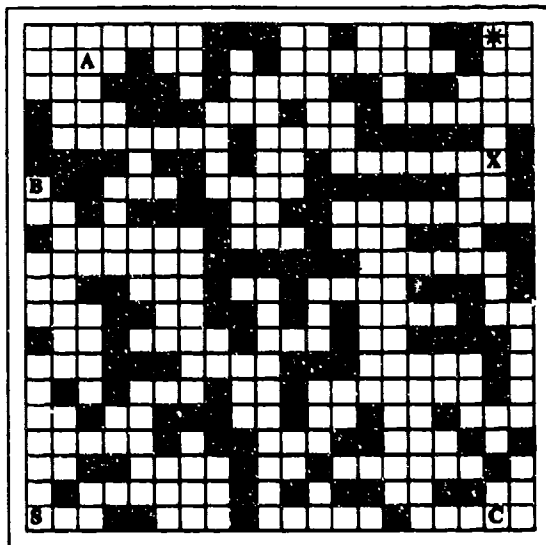


Figure 16: A maze with extra state in the form of four binary flags.

the structure of the learned state transition matrix. See Singh (1991) for current interesting work in that direction.

The third experiment is the familiar pole-balancing problem of Michie and Chambers (1968). There is no place here to discuss the enormous number of techniques which have been applied to this problem along with an equally enormous variation in details of the task formulation. The state-space of the cart is quantized at three equal levels for cart position, cart velocity, and pole angular speed. It is quantized at six equal levels for pole angle. The simulation used four real-valued state variables, yet the learner was only allowed to base its control decisions on the current quantized state. There are two actions: thrust left $10N$ and thrust right $10N$. The problem is interesting because it involves *hidden state*—the controller believes the system is Markov when in fact it is not. This is because there are many possible values for the real-valued state variables in each discretized box, and successor boxes are partially determined by these real values, which are not given to the controller. The task is defined by a reward of 100 units for every state except one absorbing state corresponding to a crash, which receives zero reward. Crashes occur if the pole angle or cart position exceed their limits. A discount factor of $\gamma = 0.999$ is used and trials start in random survivable configurations. Other parameters are ($r^{opt} = 200, \beta = 100, T_{bored} = 1, \epsilon = 10^{-3}$).

If the simulation contains no noise, or a very small amount (0.1% added to the simulated thrust),

prioritized sweeping very quickly (usually in under 1000 observations and 15 crashes) develops a policy which provides stability for approximately 100,000 cycles. With a small amount of noise (1%), stable runs of approximately 20,000 time steps are discovered after, on average, 30 crashes.

6 Heuristics to Guide Search

In all experiments to date, the optimistic estimate of the best available one-step reward, r^{opt} , has been set to an overestimate of the best reward which is actually available. However, if the human programmer knows in advance what is the best possible reward-to-go from any given state, then the resultant, more realistic, optimism does not need to experience all state-action pairs.

For example, consider the maze world. If the robot is told the location of the goal state (in all previous experiments it was not given this information), but is not told which states are blocked, then it can nevertheless compute what would be the best possible reward-to-go from a state. It could not be greater than the reward obtained from the shortest possible path to the goal. The length of the path, l , can be computed easily with the Manhattan distance metric and then the best possible reward-to-go is

$$0\gamma^1 + 0\gamma^2 + \dots + 0\gamma^{l-1} + r^{opt}\gamma^l + r^{opt}\gamma^{l+1} + \dots = \frac{r^{opt}\gamma^l}{1-\gamma} \quad (18)$$

When this optimistic heuristic is used, initial exploration is biased towards the goal, and once a path is discovered then many of the unexplored areas may be ignored. Ignoring occurs when even the most optimistic reward-to-go of a state is no greater than that of the already obtained path.

For example, Figure 17 shows the areas explored using a Manhattan heuristic when finding the optimal path from the start state at the bottom leftmost cell to the goal state at the center of the maze. The maze has 8525 states of which only 1722 needed to be explored.

For some tasks we may be satisfied to cease exploration when we have obtained a solution known to be, say, within 50% of the optimal solution. This can be achieved by using a heuristic which lies: it tells us that the best possible reward-to-go is that of a path which is twice the length of the true shortest possible path.

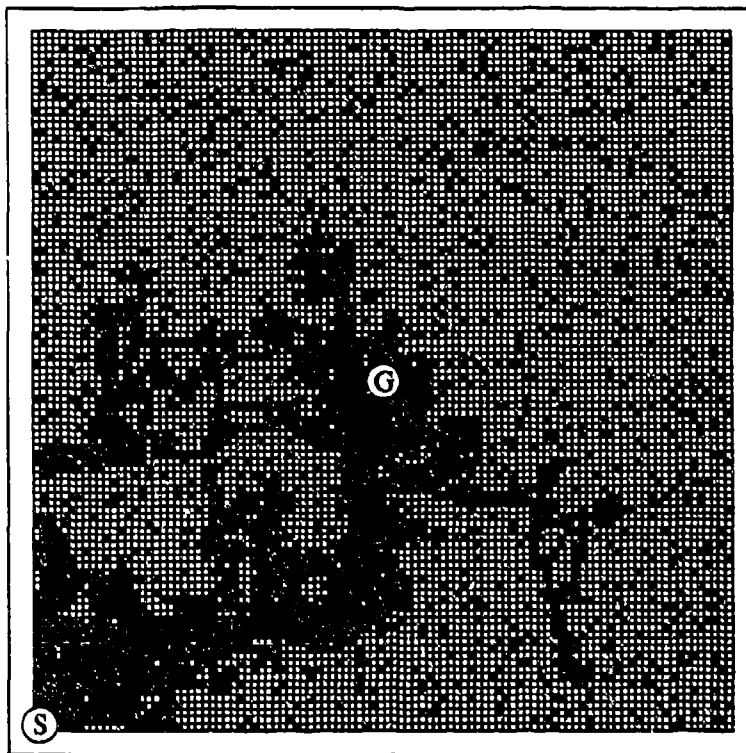


Figure 17: Dotted states are all those visited when the Manhattan heuristic was used to derive r^{opt} ($\gamma = 0.99, \beta = 10, T_{\text{bored}} = 1, \epsilon = 10^{-3}$).

7 Discussion

Generalization of the state transition model

This paper has been concerned with discrete state systems in which no prior assumptions are made about the structure of the state-space. Despite the weakness of the assumptions, we can successfully learn large stochastic tasks. However, very many problems do have extra known structure in the state-space, and it is important to consider how this knowledge can be used. By far the most common knowledge is smoothness—given two states which are in some way similar, in general their transition probabilities will be similar.

TD can also be applied to highly smooth problems using a parametric function approximator such as a neural network. This technique has recently been used successfully on a large complex problem, Backgammon, by Tesauro (1991). The discrete version of prioritized sweeping given in this paper could not be applied directly to Backgammon because the game has 10^{23} states, which

is unmanageably large by a factor of at least 10^{10} . However, a method which quantized the space of board positions, or used a more sophisticated smoothing mechanism, might conceivably be able to compute a near-optimal strategy.

We are currently developing memory-based algorithms which take advantage of local smoothness assumptions. In these investigations, state transition models are learned by memory-based function approximators (Moore and Atkeson 1992). Prioritized sweeping takes place over non-uniform tessellations of state-space, partitioned by variable resolution *kd*-trees (Moore 1991). We are also investigating the role of locally linear control rules and reward functions in such partitionings, in which instead of using Bellman's Equation (16) directly, we use local linear quadratic regulators (LQR) (see, for example, Sage and White (1977)). It is worth remembering that, if the system is sufficiently linear, LQR is an extremely powerful technique. In a pole balancer experiment in which we used local weighted regression to identify a local linear model, LQR was able to create a stable controller based on only 31 state transitions!

Other current investigations which attempt to perform generalization in conjunction with reinforcement learning are Mahadevan and Connell (1990) which investigates clustering parts of the policy, Chapman and Kaelbling (1990) which investigates automatic detection of locally relevant state variables, and Singh (1991) which considers how to automatically discover the structure in tasks such as the multiple-flags example of Figure 16.

7.1 Related work

The Dyna-Q-queue algorithm of Peng and Williams

Peng and Williams (1992) have concurrently been developing a closely related algorithm which they call Dyna-Q-queue. This conceptually similar idea was discovered independently. Where prioritized sweeping provides efficient data processing for methods which learn the state transition model, Dyna-Q-queue performs the same role for Q-learning (Watkins 1989), an algorithm which avoids building an explicit state-transition model. Dyna-Q-queue is also more careful about what it allows onto the priority queue: it only allows predecessors which have a predicted change ("interestingness"

value) greater than a significant threshold δ , whereas prioritized sweeping allows everything above a minuscule change ($\epsilon = 10^{-5}$ times the maximum reward) onto the queue. The initial experiments in Peng and Williams (1992) consist of sparse, deterministic maze worlds of several hundred cells. Performance, measured by total number of Bellman's equation processing steps before convergence, is greatly improved over conventional Dyna-Q (Sutton 1990).

Other related work

Sutton (1990) identifies reinforcement learning with asynchronous dynamic programming and introduces the same computational regime as that used for prioritized sweeping. The notion of using an optimistic heuristic to guide search goes back to the A^* tree search algorithm Nilsson (1971), which also motivated another aspect of prioritized sweeping: it too schedules nodes to be expanded according to an (albeit different) priority measure. More recently Korf (1990) gives a combination of A^* and Dynamic Programming in the LRTA* algorithm. LRTA* is, however, very different from prioritized sweeping: it concentrates all search effort in a finite-horizon set of states beyond the current actual system state. Finally, Lin (1991) has investigated a simple technique which replays, backwards, the memorized sequence of experiences which the controller has recently had. Under some circumstances this may produce some of the beneficial effects of prioritized sweeping.

8 Conclusion

Our investigation shows that prioritized sweeping can solve large state-space real time problems with which other methods have difficulty. Other benefits of the memory-based approach, described in Moore and Atkeson (1992), allow us to control forgetting in potentially changeable environments and to automatically scale state variables. Prioritized sweeping is heavily based on learning a world model and we conclude with a few words on this topic.

If a model of the world is not known to the human programmer in advance then an adaptive system is required, and there are two alternatives:

Learn a model and from this develop a control rule.

Learn a control rule without building a model.
--

Dyna and prioritized sweeping fall into the first category. Temporal differences and Q-learning fall into the second. Two motivations for *not* learning a model are (i) the interesting fact that the methods do, nevertheless, learn, and (ii) the possibility that this more accurately simulates some kinds of biological learning (Sutton and Barto 1990). However, a third advantage which is sometimes touted—that there are computational benefits in not learning a model—is, in our view, dubious. A common argument is that with the real world available to be sensed directly, why should we bother with less reliable, learned internal representations? The counterargument is that even systems acting in real time can, for every one real experience, sample millions of mental experiences from which to make decisions and improve control rules.

Consider a more colorful example. Suppose the anti-model argument was applied by a new arrival at a university campus: “I don’t need a map of the university—the university is its own map.” If the new arrival truly mistrusts the university cartographers then there might be an argument for one full exploration of the campus in order to create their own map. However, once this map has been produced, the amount of time saved overall by pausing to consult the map before traveling to each new location—rather than exhaustive or random search in the real world—is undeniably enormous.

It is justified to complain about the indiscriminate use of combinatorial search or matrix inversion prior to each supposedly real time decision. However, models need not be used in such an extravagant fashion. The prioritized sweeping algorithm is just one example of a class of algorithms which can easily operate in real time and also derive great power from a model.

Acknowledgements

Many thanks to Mary Lee, Rich Sutton and the reviewers of the paper for some very valuable comments and suggestions. Thanks also to Stefan Schaal and Satinder Singh for useful comments on an early draft. Andrew W. Moore is supported by a Postdoctoral Fellowship from SERC/NATO.

Support was also provided under Air Force Office of Scientific Research grant AFOSR-89-0500, an Alfred P. Sloan Fellowship, the W. M. Keck Foundation Associate Professorship in Biomedical Engineering, Siemens Corporation, and a National Science Foundation Presidential Young Investigator Award to Christopher G. Atkeson.

Appendix A. The random generation of a stochastic problem

Here is an algorithm to generate stochastic systems such as Figure 4 in Section 3. The parameters are: S_{nt} , the number of non-terminal states; S_t , the number of terminal states and μ_{succ} , the mean number of successors.

All states have a position within the unit square. The terminal states are generated on an equispaced circle, diameter 0.9, alternating between black and white. Non-terminal states are each positioned in a uniformly random location within the square. Then the successors of each non-terminal are selected. The number of successors is chosen randomly as $1 + X$ where X is a random variable drawn from the exponential distribution with mean $\mu_{succ} - 1$.

The choice of successors is affected by locality within the unit square. This provides a more interesting system than allowing successors to be entirely random. It was empirically noted that entirely random successors cause the long-term absorption probabilities to be very similar across most of the set of states. Locality leads to a more varied distribution.

The successors are chosen according to a simple algorithm in which they are drawn from within a slowly growing circle centered on the parent state.

If the parent state is i , and there are N_i successors, then the j th transition probability is computed by $X_j / \sum_{k=1}^{N_i} X_k$ where $\{X_1, \dots, X_{N_i}\}$ are independent random variables, uniformly distributed in the unit interval.

Once the system has been generated, a check is performed that the system is absorbing—all non-terminals can eventually reach at least one terminal state. If not, entirely new systems are randomly generated until an absorbing Markov system is obtained.

References

Barto, A. G. and Singh, S. P. (1990). On the Computational Economics of Reinforcement Learning. In Touretzky, D. S., editor, *Connectionist Models: Proceedings of the 1990 Summer School*. Morgan Kaufmann.

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1989). Learning and Sequential Decision Making. COINS Technical Report 89-95, University of Massachusetts at Amherst.

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1991). Real-time Learning and Control using Asynchronous Dynamic Programming. Technical Report 91-57, University of Massachusetts at Amherst.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Berry, D. A. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall.

Bertsekas, D. P. and Tsitsiklis, J. N. (1989). *Parallel and Distributed Computation*. Prentice Hall.

Chapman, D. and Kaelbling, L. P. (1990). Learning from Delayed Reinforcement In a Complex Domain. Technical Report No. TR-90-11, Teleos Research.

Christiansen, A. D., Mason, M. T., and Mitchell, T. M. (1990). Learning Reliable Manipulation Strategies without Initial Physical Models. In *IEEE Conference on Robotics and Automation*, pages 1224-1230.

Dayan, P. (1992). The Convergence of $TD(\lambda)$ for General λ . *Machine Learning*, 8(3).

Kaelbling, L. P. (1990). Learning in Embedded Systems. PhD. Thesis; Technical Report No. TR-90-04, Stanford University, Department of Computer Science.

Knuth, D. E. (1973). *Sorting and Searching*. Addison Wesley.

Korf, R. E. (1990). Real-Time Heuristic Search. *Artificial Intelligence*, 42.

Lin, L. J. (1991). Programming Robots using Reinforcement Learning and Teaching. In *Proceedings of the Ninth International conference on Artificial Intelligence (AAAI-91)*. MIT Press.

Mahadevan, S. and Connell, J. (1990). Automatic Programming of Behavior-based Robots using Reinforcement Learning. Technical Report, IBM T. J. Watson Research Center, NY 10598.

Michie, D. and Chambers, R. A. (1968). BOXES: An Experiment in Adaptive Control. In Dale, E. and Michie, D., editors, *Machine Intelligence 2*. Oliver and Boyd.

Moore, A. W. and Atkeson, C. G. (1992). Memory-based Function Approximators for Learning Control. In preparation.

Moore, A. W. (1991). Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-valued State-spaces. In Birnbaum, L. and Collins, G., editors, *Machine Learning: Proceedings of the Eighth International Workshop*. Morgan Kaufman.

Nilsson, N. J. (1971). *Problem-solving Methods in Artificial Intelligence*. McGraw Hill.

Peng, J. and Williams, R. J. (1992). Efficient Search Control in Dyna. College of Computer Science, Northeastern University.

Sage, A. P. and White, C. C. (1977). *Optimum Systems Control*. Prentice Hall.

Samuel, A. L. (1959). Some Studies in Machine Learning using the Game of Checkers. *IBM Journal on Research and Development*, 3. Reprinted in Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*, McGraw-Hill, 1963.

Sato, M., Abe, K., and Takeda, H. (1988). Learning Control of Finite Markov Chains with an Explicit Trade-off Between Estimation and Control. *IEEE Trans. on Systems, Man, and Cybernetics*, 18(5):667-684.

Singh, S. P. (1991). Transfer of learning across compositions of sequential tasks. In Birnbaum, I. and Collins, G., editors, *Machine Learning: Proceedings of the Eighth International Workshop*. Morgan Kaufman.

Stanfill, C. and Waltz, D. (1986). Towards Memory-Based Reasoning. *Communications of the ACM*, 29(12):1213-1228, December.

Sutton, R. S. and Barto, A. G. (1990). Time-Derivative Models of Pavlovian Reinforcement. In Gabriel, M. and Moore, J., editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 497-537. MIT Press.

Sutton, R. S. (1984). Temporal Credit Assignment in Reinforcement Learning. Phd. thesis, University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3:9-44.

Sutton, R. S. (1990). Integrated Architecture for Learning, Planning, and Reacting Based on

Approximating Dynamic Programming. In *Proceedings of the 7th International Conference on Machine Learning*. Morgan Kaufman.

Tesauro, G. J. (1991). Practical Issues in Temporal Difference Learning. Report RC 17223 (76307), IBM T. J. Watson Research Center, NY.

Thrun, S. B. and Möller, K. (1992). Active Exploration in Dynamic Environments. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*. Morgan Kaufmann.

Watkins, C. J. C. H. (1989). Learning from Delayed Rewards. PhD. Thesis, King's College, University of Cambridge.