Memory-Based Learning for Control

A. W. Moore, C. G. Atkeson and S. A. Schaal

CMU–RI–TR–95–18

A. W. Moore
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

C. G. Atkeson and S. Schaal
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

April 1995

Contents

Memory-Based Learning for Control

Andrew W. Moore
Carnegie Mellon Univ.
5000 Forbes Ave
Pittsburgh, PA 15213
awm@cs.cmu.edu

Christopher Atkeson and Stefan Schaal
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA 30332-0280
(cga,sschaal)@cc.gatech.edu

Abstract

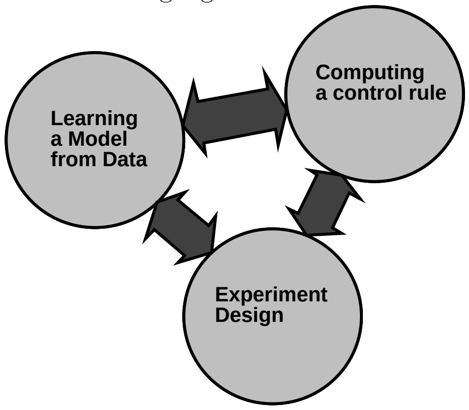
The central thesis of this article is that memory-based methods provide natural and powerful mechanisms for high-autonomy learning control. This paper takes the form of a survey of the ways in which memory-based methods can and have been applied to control tasks, with an emphasis on tasks in robotics and manufacturing. We explain the various forms that control tasks can take, and how this impacts on the choice of learning algorithm. We show a progression of five increasingly more complex algorithms which are applicable to increasingly more complex kinds of control tasks. We examine their empirical behavior on robotic and industrial tasks. The final section discusses the interesting impact that explicitly remembering all previous experiences has on the problem of learning control.

1 Learning Control

The necessity for self improvement in control systems is becoming more apparent as fields such as robotics, factory automation and autonomous vehicles are being impeded by the complexity of inventing and programming satisfactory control laws. We are interested in applying memory-based learning to control tasks in robotics and manufacturing. These tasks often involve decisions based on streams of information from sensors and actuators, where data is not to be wasted, but can be relatively plentiful. This paper discusses the opportunities that arise if everything that happens to a system in its lifetime is remembered.

One of the important distinctions that became clear to us in the process of implementing learning controllers is the difference between representational tools, such as lookup tables, neural networks, or structured representations, and what we will call learning paradigms, which define what the representation is used for, where training data comes from, how the training data is used to modify the representation, whether exploratory actions are performed, and other related issues. It is difficult to evaluate a representational tool independently of the paradigm in which it is used, and vice versa. A successful robot learning algorithm typically is composed of sophisticated representational tools and sophisticated learning paradigms.

There are three readily identifiable components of a learning control system, depicted in the following figure.



The control rule computation relies on the empirical model to make its decisions. Building the empirical model requires data, and the experiment design component decides which data is most valuable to obtain. And then, in turn, the control rule may choose actions in order to help obtain that data.

This paper discusses all three components as they are used in a number of learning control algorithms. The aim of this paper is to survey the implications of *memory-based* approaches for the performance of a controller. In memory-based learning, experiences are explicitly remembered, and predictions and generalizations are performed on line in real time by building a local model to answer any particular query (an input for which the function's output is desired). Our approach is to model complex functions using simple local models. One benefit of memory-based modeling is that it avoids the difficult problem of finding an appropriate structure for a global model, since there is no global model.

A key idea in memory-based modeling is to form a training set for the local model after the query that must 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. We form a local 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 for that query. After answering the query, the local model is discarded and a new local model is created to answer the next query. This leads to another benefit of memory-based modeling for control: the structure of the local model and the structural parameters do not have to be chosen until after the data is collected. In fact, choices are forced only when a query arrives, and any choices can be changed when the next query arrives.

The memory-based architecture can represent nonlinear functions, yet has simple training rules with a single global optimum for building a local model in response to a query. This allows complex nonlinear models to be identified (trained) quickly. 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. Fast training makes continuous learning from a stream of new input data possible. It is true that memory-based learning transfers all the computational load onto the lookup process, but our experience is that the linear parameter estimation process during lookup is still fast enough for real time robot learning.

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 fact, performing a cross validation in a memory-based model is no more expensive than processing a single query. Cheap cross validation makes search for model parameters routine, and we have explored procedures that take advantage of this.

Robots should be able to learn multiple tasks. Unfortunately, changing the data distribution causes problems for parametric models. Since black box models such as neural networks do not have exactly the correct model structure for a task, the distribution of the data affects the values of the parameters. When the data distribution changes (because the robot is doing something slightly different) the parameters also change, leading to degraded performance on the original task. Because memory-based models store the actual data and avoid retaining any constructs based on the data, they do not suffer from this type of negative interference.

We have extended the memory-based approach to give information about the reliability of the predictions and local linearizations generated, based on the local density and distribution of the data and an estimate of the local variance. This allows the robot to monitor its own skill level, protect itself from its own ignorance by designing robust policies, and guide its exploratory behavior.

Another attractive feature of memory-based learning is that there are explicit parameters to control smoothing, outlier rejection, forgetting, and other processes. The modeling process is easy to understand, and therefore easy to adjust or control.

We will see how the explicit memories can speed up convergence and improve the robustness and autonomy of optimization and control algorithms. It is very frustrating to watch a robot repeat the same mistake over and over, with only a slight improvement on each attempt. The goal of the learning algorithms described here is to have rapid improvement in performance.

We will not review memory-based function approximation here, but refer the reader to the companion papers in this collection. Here, we use the terminology and notation of [Atkeson *et al.*, 1995].

2 Learning control with empirical models

In this paper we will deal with learning control problems in which, on each cycle, the controller first senses its current state, a vector $\mathbf{x}$. A task description specifies the desired behavior, a vector $\mathbf{y}_{\mathrm{des}}$. The controller must then choose an action, which is another vector, $\mathbf{u}$. Having performed the action, the controller observes the actual resulting behavior $\mathbf{y}_{\mathrm{actual}}$ and may use this data to improve its subsequent performance.

Several relationships could be acquired by a learning controller. The first is the forward model (**State** $\times$ **Action** $\rightarrow$ **Behavior**) [Miller, 1989, Mel, 1989, Moore, 1990, Jordan and Rumelhart, 1992].

$$\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{u})$$

This allows one to predict the effects of various actions but initially appears unattractive because it does not proscribe the correct action to take.

An alternative is the *policy* (**State** $\rightarrow$ **Action**) [Michie and Chambers, 1968, Barto *et al.*, 1989, Watkins, 1989].

$$\mathbf{u} = \boldsymbol{\pi}(\mathbf{x})$$

The policy may initially appear the most attractive mapping because it provides exactly what the controller needs: a mapping from the state we have observed to the action we should apply. Its disadvantage is that if learned on its own, then a strategy for updating the policy requires extra fore-knowledge of the structure of the world: if we apply an action which gives substandard behavior, we need to know which direction to alter the mapping so that next time the adjusted action will produce improved behavior. This problem is alleviated if the policy is learned in conjunction with a more easily updatable mapping, such as the forward world model [Jordan and Rumelhart, 1992, Sutton, 1990a], and perhaps also a value function (see Section 7).

The *inverse model* (**State** $\times$ **Behavior** $\rightarrow$ **Action**) [Atkeson, 1989, Miller, 1989], is of the form

$$\mathbf{u} = \mathbf{f}^{-1}(\mathbf{x}, \mathbf{y}).$$

A superior feature of this scheme over a policy is that we are adding objective observations of the world, rather than adjustments to a task-specific map. For example, if a Billiards-playing robot is learning to sink balls in the far left hole, and it accidentally sinks it in the far right hole, then despite the immediate failure, the robot has the consolation that should it in future be asked to sink into the far right, it will have remembered the means to satisfy the request.

This paper is organized by types of control tasks. In the next few sections we will examine control tasks of increasing complexity. For each task-type we will show memory-based algorithms for learning control, and for several task-types we also provide details of robotic implementations. The progression of control tasks is given in the following table.

Task	Task Specification	Goal	Example	Sec.
Temporally Independent Task	$\mathbf{y}_{\text{des}}$	Choose $\mathbf{u}$ such that $E[\mathbf{y}] = \mathbf{y}_{\text{des}}$.	Billiards, throwing, setpoint-based process control	3
Dead-beat control	$\mathbf{x}_{\text{des}}$ or trajectory $\{\mathbf{x}_{\text{des}}(t)\}$	Choose $\mathbf{u}(t)$ such that $E[\mathbf{x}(t+1)] = \mathbf{x}_{\text{des}}(t+1)$.	Ball bouncing, some process control	4
Dynamic Control	Matrices $\mathbf{Q}$ and $\mathbf{R}$, $\mathbf{x}_{\text{des}}$ or trajectory $\{\mathbf{x}_{\text{des}}(t)\}$	Minimize future sum Cost $= \sum_{t=0}^{\infty} \left(\mathbf{d}(t)^T \mathbf{Q} \mathbf{d}(t) + \mathbf{u}(t)^T \mathbf{R} \mathbf{u}(t)\right)$ where $\mathbf{d}(t) = \mathbf{x}(t) - \mathbf{x}_{\text{des}}(t)$	Trajectory tracking, regulation	5
Dynamic Regulation, setpoint unknown	Matrices $\mathbf{Q}$ and $\mathbf{R}$	Find a stable setpoint such that $\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))$	Juggling, Pole-balancing	6
General Non-linear Optimal Control	Cost function $Cost(\mathbf{x}, \mathbf{u})$.	Find a control policy to minimize the sum of future costs.	(All previous, plus) Car-on-hill, Packaging, Plant setup and shutdown	7

3 Temporally Independent Decisions

In the simplest class of task we will consider, the environment provides a behavior $\mathbf{y}(t)$ as a function of an action $\mathbf{u}(t)$ which we can choose, a state

$\mathbf{x}(t)$ we can observe but not choose, and random noise.

$$\mathbf{y}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), \text{noise}(t))$$

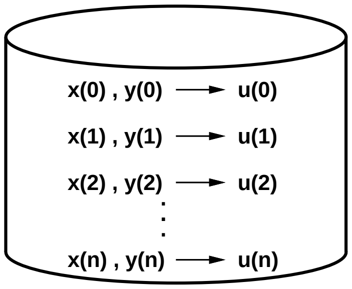
The task is to choose $\mathbf{u}(t)$ to achieve $E[\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))] = \mathbf{y}_{\text{des}}$. The function $\mathbf{f}$ is an a-priori unknown. An important assumption (which we will relax later) is that $\mathbf{x}(t+1)$ is independent of $\mathbf{u}(t)$, so there is no opportunity to choose suboptimal actions in the short-term to improve performance in the long-term.

It is considerably easier to reason about the optimality of, and compute the optimal controls for, a temporally independent problem, because all decision making is concerned entirely with the outcome of the immediate action. There is no need to consider the effects of the current action on the future performance in later time steps.

Despite this relative ease, the problems still form an interesting class of learning tasks. They are different from conventional supervised learning, because a decision must be made at each time step, and that decision both depends on inferences from earlier data and affects the future data available to future decisions.

Inverse-model based learning control

The learned inverse model can provide a conceptually simple controller for some temporally independent or dead-beat control problems. The memory-base is arranged so that the input vectors of each datapoint are the concatenation of state and behavior vectors. The corresponding output is the action needed to produce the given behavior from the given state.



With this memory-base there is a very simple algorithm for simultaneously learning control and gathering data in the important parts of the action space.

Algorithm: Inv-Mod

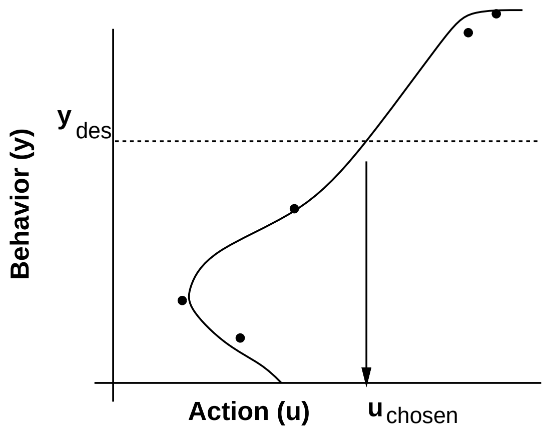


Figure 1: Given the initial data (black dots) the inverse model (read from the y axis to the x axis) can be used to predict the action that will achieve the desired behavior.

1. Observe state $\mathbf{x}$ and goal behavior $\mathbf{y}_{\mathrm{des}}$.
2. Choose $\mathbf{u} = \hat{\mathbf{f}}^{-1}(\mathbf{x}, \mathbf{y}_{\mathrm{des}})$. If there are insufficient data in the memory-base to make a prediction, choose an experimental action.
3. Perform action $\mathbf{u}$ in the real world.
4. Observe actual behavior, $\mathbf{y}$.
5. Update the memory with $(\mathbf{x}, \mathbf{y}) \to \mathbf{u}$.

The strength of this algorithm in conjunction with a memory based learner is demonstrated in Figure 1. The learning is aggressive: during repeated attempts to achieve the same goal behavior, the action which is applied is not an incrementally adjusted version of the previous action, but is instead the action which the memory and the memory-based learner predicts will directly achieve the required behavior. In this simple monotonic function the sequence of actions which are chosen by the Inv-Mod algorithm are closely related to the Secant method [Conte and De Boor, 1980] for numerically finding the zero of function by bisecting the line between the two most recent approximation by the $y = 0$ axis. See [Ortega and Rheinboldt, 1970] for a good discussion of the multidimensional generalization of Secant.

If the learning process begins with an initial error E_0 in the action choice, and we wish to reduce this error to E_0/K, then the number of learning steps is $O(\log\log K)$: subject to benign conditions, this learner jumps to actions very close to the ideal action very quickly. Inv-Mod, trained initially with a

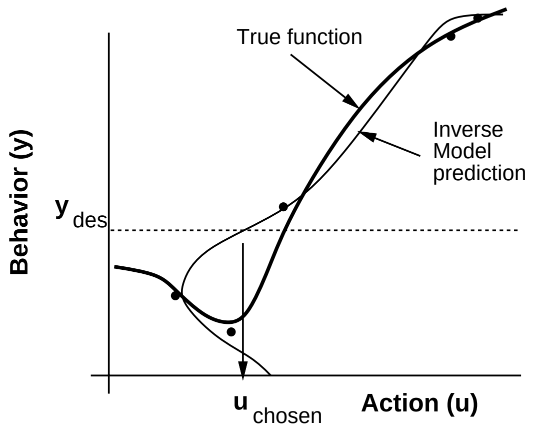


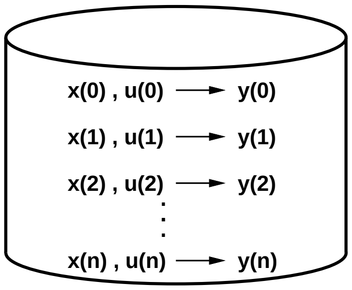
Figure 2: The true relation (shown as the thick black line) is non-monotonic. When a behavior is desired at the shown value, the action that is suggested produces a behavior that differs from the desired one. Worse, the new datapoint that is added (at the intersection of the black line and the vertical arrow) will not change the inverse model near $\mathbf{y}_{des}$, and the same mistake will be repeated indefinitely.

feedback learner, has been used with locally weighted regression by [Atkeson, 1989].

A commonly observed problem with the inverse model is that if the vector space of actions has a different dimensionality than that of behaviors then the inverse model is not well defined. More insidiously, even if they are the same dimensionality then if the mapping is not 1-1 (which implies globally continuous and monotonic), or if there are some misleading noisy observations, then learning can become stuck in permanent pockets of inaccuracy which are not reduced with experience. Figure 2 illustrates a problem where a non-monotonic relation between actions and behaviors is misinterpreted by the inverse model. Even if the inverse model had interpreted the data correctly the averaging it would have done on $\mathbf{u}$ would have led to incorrect actions [Moore, 1991a, Jordan and Rumelhart, 1992].

Forward-model based learning control

We now arrange the memory-base so that the input vectors of each datapoint are the concatenation of state and action vectors. The corresponding output is the actual behavior that was observed when the state-action pair was executed in the real world.



To use this model for control requires more than a simple lookup. Actions are chosen by on line numerical inversion of a memory-based forward model. This is illustrated by the following algorithm

Algorithm: For-Mod

1. Observe state $\mathbf{x}$ and goal behavior $\mathbf{y}_{\text{des}}$.
2. Perform numerical inversion: search among a series of candidate actions, $\mathbf{u}_1, \mathbf{u}_2, \ldots$.

   $$\hat{\mathbf{y}}_i := \hat{\mathbf{f}}(\mathbf{x}, \mathbf{u}_i)$$

   to find an action $\mathbf{u}_k$ for which $\hat{\mathbf{y}}_i = \mathbf{y}_{\text{des}}$.
3. If no action was found within the allotted number of candidate actions (or allotted real time) then use an experimental, or default, action.
4. Observe actual behavior, $\mathbf{y}$, and update the memory with $(\mathbf{x}, \mathbf{u}_k \to \mathbf{y})$.

Step 2 requires that a set of actions are searched to find one that predicts the desired output. This computation is exactly equivalent to numerical root finding over the empirical model. A number of root-finding schemes are applicable, with desirability depending on the dimensionality of the actions, the complexity of the function and the amount of real-time available with which to perform the search.

- Generate all available actions sampled from a uniform grid over action space. Use the action which is predicted to produce the closest behavior to $\mathbf{y}_{\text{des}}$.
- Generate random actions, and again use the action which is predicted to produce the closest behavior to $\mathbf{y}_{\text{des}}$.

- Perform a steepest-ascent search from an initial candidate action towards an action that will give the desired output. Finding the local gradient of the empirical model is easy if locally weighted regression is used. Part of the computation of the locally weighted regression model forms the local linear map, and so it is available for free. We may write the prediction local to $\mathbf{x}$ and $\mathbf{u}$ as

  $$\hat{\mathbf{f}}(\mathbf{x} + \boldsymbol{\delta}\mathbf{x}, \mathbf{u} + \boldsymbol{\delta}\mathbf{u}) = \mathbf{c} + \mathbf{A}\boldsymbol{\delta}\mathbf{x} + \mathbf{B}\boldsymbol{\delta}\mathbf{u} + \text{2nd order terms}$$

  where $\mathbf{c}$ is a vector and $\mathbf{A}$ and $\mathbf{B}$ are matrices obtained from the regression, such that

  $$\mathbf{c} = \hat{\mathbf{f}}(\mathbf{x}, \mathbf{u}) \quad \mathbf{A}_{ij} = \frac{\partial \hat{f}_i}{\partial x_j} \quad \mathbf{B}_{ij} = \frac{\partial \hat{f}_i}{\partial u_j} \tag{1}$$

- Use Newton's method to iterate towards an action with the desired output. If $\mathbf{u}_k$ is an approximate solution, Newton's method gives $\mathbf{u}_{k+1}$ as a better solution where

  $$\mathbf{u}_{k+1} = \mathbf{u}_k + \mathbf{B}^{-1}(\mathbf{y}_{\text{des}} - \mathbf{c})$$

  with $\mathbf{B}$ and $\mathbf{c}$ as defined in Equation 1. Newton's method is less stable in general, but if a good approximate solution is available (perhaps from one of the earlier methods) it produces a very accurate action in only two or three iterations.

If the partial derivatives matrix $\mathbf{B}$ is singular, or the action space and state space differ in dimensionality, then robust matrix techniques based on the pseudo-inverse can be applied to this procedure. The forward model can also be used to minimize a criteria that penalizes large commands as well as behavior errors, which makes this search more robust:

$$\text{Cost} = (\mathbf{y}_{\text{des}} - \mathbf{c})^T \mathbf{Q} (\mathbf{y}_{\text{des}} - \mathbf{c}) + \mathbf{u}^T \mathbf{R} \mathbf{u}$$

The matrices $\mathbf{Q}$ and $\mathbf{R}$ allow control of which components of the error are most important.

This learning control algorithm does not require a human trainer, nor does it require external guidance as to when it should experiment. If an action is wrongly predicted to succeed, then the algorithm will apply the action, and the resulting new datapoint will prevent the same mistake in future predictions.

Combining Forward and Inverse models
It is easy to combine the previous two algorithms so that the inverse model can provide a good initial start-point for the search. The initial estimate of a good action action is generated by

$$\mathbf{u}_0 = \hat{\mathbf{f}}^{-1}(\mathbf{x}, \mathbf{y}_{\mathrm{des}})$$

Then $\mathbf{u}_0$ can be evaluated using a memory-based forward model with the same data.

$$\hat{\mathbf{y}} = \hat{\mathbf{f}}(\mathbf{x}, \mathbf{u}_0)$$

Provided $\hat{\mathbf{y}}$ is close to $\mathbf{y}_{\mathrm{des}}$, Newton's method can then be used for further refinement.

Experiment design for temporally independent learning
A nice feature of these algorithms is that in normal operation they perform their own experimental design. The experiments are chosen greedily at the exact points where the desired output is predicted to be, which for the forward model is guaranteed to provide a useful piece of data.

In the early stages of learning, however, there may be no action that is predicted to give the desired behavior. A simple experiment design strategy is to choose actions at random. Far more effective, however, is to choose datapoints which, given the uncertainty inherent in the prediction, are considered most likely to achieve the desired behavior. This can considerably reduce the exploration required [Moore, 1991a, Schaal and Atkeson, 1994b, Cohn *et al.*, 1995].

Example (robotic): Billiards
Some experiments were performed with the billiards robot shown in Figure 3. See [Moore, 1992, Moore *et al.*, 1992] for more details of the experiment. The equipment consists of a small ($1.5m \times 0.75m$) pool table, a spring actuated cue with a rotary joint under the control of a stepper motor, and two cameras attached to a Datacube image processing system.

All sensing is visual: one camera looks along the cue stick and the other looks down at the table. The cue stick swivels around the cue ball, which, in these experiments, has to start each shot at the same position. A shot proceeds as follows:

1. At the start of each attempt the object ball (the ball which we will try to sink in a pocket) is placed at a random position in the half of

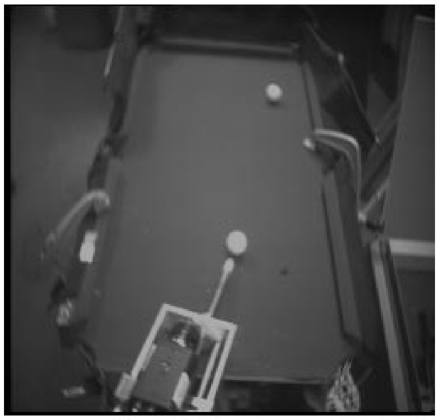


Figure 3: The billiards robot. In the foreground is the cue stick which attempts to sink balls in the far pockets.

the table opposite the cue stick. This random position is selected by the computer in order to avoid human bias.

2. The camera above the table obtains the centroid image coordinates of the object ball $(x^{\text{above}}_{\text{object}}, y^{\text{above}}_{\text{object}})$, which constitute the state $\mathbf{x}$.

3. The controller then uses the forward-and-inverse algorithm to find an action, $\mathbf{u}$, that is predicted to sink the object ball into the nearer of the two pockets at the far end of the table. The action is specified by what we wish the view from the cue to be just prior to shooting. Figure 4 shows a view from the cue camera during this process. The cue swivels until the centroid of the object ball's image (shown by the vertical line) coincides with the chosen action, $x^{\text{cue}}_{\text{object}}$, shown by the cross.

4. The shot is then performed and observed by the overhead camera. The image after a shot, overlaid with the tracking of both balls, is shown in Figure 5. The behavior is defined as the cushion and position on the cushion with which the object ball first collides. In Figure 5 it is the point b.

5. Independent of success or failure, the memory-base is updated with the new observation $(x^{\text{above}}_{\text{object}}, y^{\text{above}}_{\text{object}}, x^{\text{cue}}_{\text{object}}) \rightarrow b$.

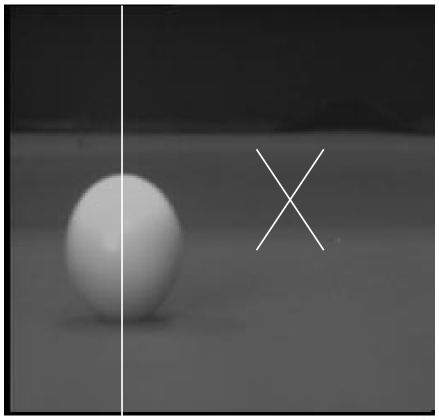


Figure 4: The view from the cue camera during aiming

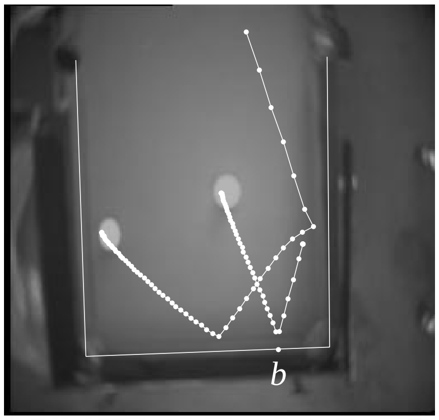


Figure 5: Tracking the shot with the overhead camera

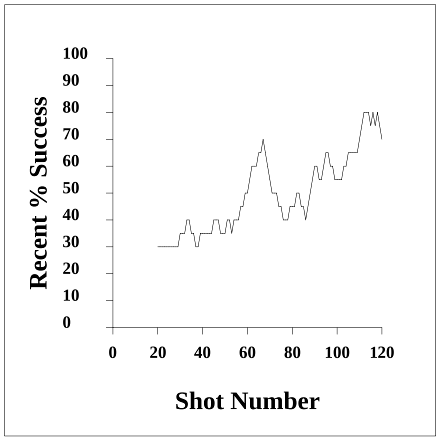


Figure 6: Frequency of successes versus control cycle for the billiards task. The number of successes, averaged over the twenty previous shots, is shown.

As time progresses, the database of experiences increases, hopefully converging to expertise in the two-dimensional manifold of state-space corresponding to sinking balls placed in arbitrary positions. Before learning begins there is no explicit knowledge or calibration of the robot, pool table, or cameras, beyond having the object ball in view of the overhead camera, and the assumption that the relationship between state, action and behavior is reasonably repeatable.

In this experiment the empirical model was locally weighted regression using outlier removal and cross validation for choosing the kernel width. Inverse and forward models were used together; the forward model was searched with steepest ascent. Early experiments (when no success was predicted) were uncertainty-based [Moore, 1991a]. After 100 experiences, control choice running on a Sun-4 was taking 0.8 seconds.

A graph of the number of successes against trial number (Figure 6) shows the performance of the robot against time. Sinking the ball requires better than 1% accuracy in the choice of action, the world contains discontinuities and there are random outliers in the data due to visual tracking errors, and so it is encouraging that within less than 100 experiences the robot had reached a 75% success rate. This is substantially better than the authors can achieve and given the limited repeatability of the robot, perhaps close to as high a success rate as possible.

This experiment demonstrates several important points. The primary point is accuracy. The cue-action must be extremely precise for success, and it would have been inadequate to use a function approximator which merely gave the "general trends of the data". The second point is the non-uniformity of the data. Although the function being learned is only 3 inputs $\to$ 1 output, it is perhaps surprising that it achieved the necessary accuracy in only 100 datapoints. The reason is the aggressive non-uniformity of the the data distribution—almost all clustered around state-action pairs which get the ball in or close to a pocket.

4 Dead-beat control

A more complex class of learning control tasks occur when the assumption of temporal independence is removed: $\mathbf{x}(t+1)$ may now be influenced by $\mathbf{x}(t)$. Indeed, a common case is

$$\mathbf{y}(t) = \mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))$$

and the very behavior we are controlling is the system state. The task may be to regulate the state to a predefined desired value called a *setpoint* $\mathbf{x}_{\text{des}}$ or a trajectory of states

$$\mathbf{x}_{\text{des}}(1), \mathbf{x}_{\text{des}}(2), \mathbf{x}_{\text{des}}(3) \ldots$$

The clearest approach to achieving the trajectory is one-step *dead-beat* control, in which each action is chosen to (in expectation) cause the immediate next state to be the desired next state. Assuming the next state is always attainable in one step, the action may again be chosen with only the immediate decision in mind, and not paying attention to the needs of future decisions and the methods of Section 3 can be used directly. The ball-bouncing task of [Aboaf *et al.*, 1989] provides an example.

5 Dynamic Control

In many dynamic control problems, it is not possible to return to the desired setpoint or trajectory in one step: an attempt to do so would require actions of infeasibly large magnitude or cause an instability. Dead-beat control will fail on non-minimum phase systems, of which the pole on a cart is one example. In these systems, one must move away from the goal in order to approach it later.

Commonly, a controller performs more robustly and effectively if it is encouraged to use smaller magnitude actions and return to the correct trajectory in a larger number of steps.

This idea is posed precisely in the language of LQ (linear quadratic) regulation, in which a long term cost function (over many time steps) is to be minimized:

$$\text{Cost} = \sum_{t=0}^{\infty} \Big((\mathbf{x}(t) - \mathbf{x}_{\text{des}})^T \mathbf{Q} (\mathbf{x}(t) - \mathbf{x}_{\text{des}}) + \mathbf{u}(t)^T \mathbf{R} \mathbf{u}(t) \Big)$$

where $\mathbf{Q}$ is a positive definite and $\mathbf{R}$ a positive semi-definite matrix. If, for example, $\mathbf{Q}$ and $\mathbf{R}$ were set to identity matrices, then the sum of squared deviation from desired performance would be penalized. The human supervisor can specify the various magnitudes of importance of the various state and action components by suitable adjustments to $\mathbf{Q}$ and $\mathbf{R}$.

Not using dead-beat policies implies some amount of lookahead. LQR control assumes a time invariant task and performs an infinite amount of lookahead. Predictive or Receding Horizon control design techniques look N steps ahead. These techniques will allow larger state errors in order to reduce the size of the control signals. The exact tradeoff is set by choosing $\mathbf{Q}$ and $\mathbf{R}$ in the optimization criteria. At this point it is not clear what balance of $\mathbf{Q}$ and $\mathbf{R}$ is best for learning.

Once in this form, we can take advantage of the locally linear state-transition function provided by locally weighted regression.

$$\mathbf{x}(t+1) = \mathbf{c} + \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$$

The optimal action can be obtained by solution of a matrix system called the Ricatti equation [Sage and White, 1977]. If we translate the coordinate system we can consider the problem as one in which the goal is to minimize

$$\text{Cost} = \sum_{t=0}^{\infty} \Big(\mathbf{x}(t)^T \mathbf{Q} \mathbf{x}(t) + \mathbf{u}(t)^T \mathbf{R} \mathbf{u}(t) \Big)$$

subject to the linear system $\mathbf{x}(t+1) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$ Then, optimal control theory gives us, that subject to our assumption of local linearity, the long term cost starting from state $\mathbf{x}$ is $\mathbf{x}^T\mathbf{P}\mathbf{x}$ where $\mathbf{P}$ is obtained by running the following iteration to convergence.

1. $\mathbf{P} := \mathbf{Q}$
2. $\mathbf{P} := \mathbf{Q} + \mathbf{A}^T\mathbf{P}[\mathbf{I} + \mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\mathbf{P}]^{-1}\mathbf{A}$
3. Unless converged, goto Step 2.

The same theory gives the optimal action $\mathbf{u}$ as a simple gain matrix $\mathbf{K}$

$$\mathbf{u} = -(\mathbf{R} + \mathbf{B}^T\mathbf{P}\mathbf{B})^{-1}\mathbf{B}^T\mathbf{P}\mathbf{A}\mathbf{x} = -\mathbf{K}\mathbf{x}$$

Policies based on LQR designs will not be useful if the task requires operation outside a locally linear region. The LQR controller may actually be unstable. For example, the following system

$$x_{k+1} = 2x_k + u_k + x_k^2 u_k \tag{2}$$

has a local linear model at the origin ($x = 0$) of $\mathbf{A} = 2$ and $\mathbf{B} = 1$. For the optimization criteria $\mathbf{Q} = 1$ and $\mathbf{R} = 1$, $\mathbf{K} = 1.618$. For a goal of moving to the origin, this linear control law is unstable for x larger than 0.95. In cases where the nonlinearity is significant the full dynamic programming based policy design approach may be used (Section 7).

6 Unknown optimal setpoints

The LQR method combined with LWR empirical modeling is a direct application of conventional control theory. The learning task is considerably harder if the desired trajectory is unknown in advance, and instead must itself be optimized in order to achieve some higher level task description.

In the general case, it is necessary to learn an optimal trajectory in addition to learning to optimally track the trajectory. That case we leave until the next section. A special case is when the task is a *regulation* task, which means that the optimal solution involves maintaining the system at a fixed setpoint. If the location of the setpoint were known in advance we could use conventional LQR in combination with our model. If the location of the setpoint is unknown, we have an additional part of the task to learn, which can be addressed by the *shifting setpoint algorithm* [Schaal and Atkeson, 1994a].

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.

Experiment Design with Shifting Setpoints

The major ingredient of the SSA is a statistical self-monitoring process.

Whenever the current location in input space $\mathbf{x}$ has obtained a sufficient amount of experiences such that a measure of confidence rises above a threshold, the setpoints are shifted in the direction of the goal until the confidence falls below a minimum confidence level. At this new setpoint location, the learning system collects new experiences. The shifting process is repeated until the goal is reached. In this way, the SSA builds a narrow tube of data support in which it knows the world. This data builds the basis for the first success of the regulator controller. Subsequently, the learned model of the world can be used for more sophisticated control algorithms for planning or further exploration.

Example (robotic): Devil Sticking
The SSA method was tested on a juggling task known as *devil sticking*. More details may be found in [Schaal and Atkeson, 1994b, Schaal and Atkeson, 1994a]. A center stick is batted back and forth between two handsticks. Figure 7 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. The task state is the predicted location of the center stick when it hits the hand stick held in a nominal position. Standard ballistics equations for the flight of the center stick are used to map flight trajectory measurements into a task state. The dynamics of throwing the devilstick are parameterized by 5 state and 5 action variables, 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 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 that results. The SSA algorithm was applied in the following steps.

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 is collected somewhere in state-action space of each hand. An abstract illustration for this is given in Figure 8a.

(a)

(b)

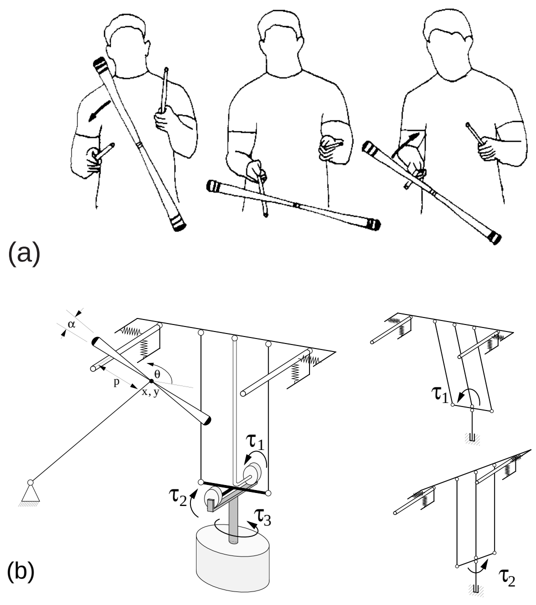


Figure 7: (a) an illustration of devil sticking, (b) sketch of our devil sticking robot: the flow of force from each motor into the robot is indicated by different shadings of the robot links, and a position change due to an application of motor 1 or motor 2, respectively, is indicated in the small sketches.

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. The linear model is estimated by LWR. 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 (Figure 8b).

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 (Figure 8b).

4. The SSA repeats itself by collecting data in the new regions of the workspace until the setpoints can be shifted again (Figure 8c). The procedure terminates by reaching the goal, leaving a ridge of data in space (Figure 8d).

The LQ controllers play a crucial role for devil sticking. Although we

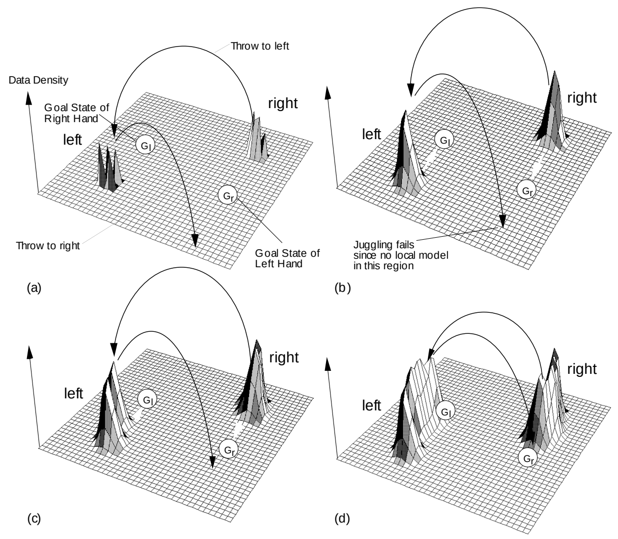


Figure 8: Abstract illustration how the SSA algorithm collects data in space: (a) sparse data after the first few hits; (b) high local data density due to local control in this region; (c) increased data density on the way to the goals due to shifting the setpoints; (d) ridge of data density after the goal was reached.

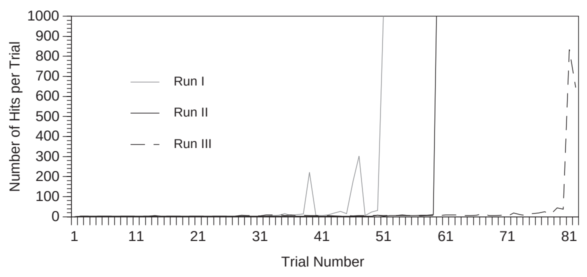


Figure 9: Learning curves of devil sticking for three runs.

statistically exploit data rather thoroughly, it is nevertheless hard to build good local linear models in the high dimensional forward models, particularly at the beginning of learning. LQ control has useful robustness even if the underlying linear models are still rather imprecise.

The SSA was tested in a noise corrupted simulation and on the real robot. Learning curves of the real robot are given in Figure 9. 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 breakthrough occurs and the robot rarely loses the devilstick after that. At this time, about 400 data points have been collected in memory. The real robot's learning performance is qualitatively the same, only that due to the stronger nonlinearities and unknown noise sources learning takes more trials to accomplish a steady juggling pattern. Peak performance of the robot was more than 2000 consecutive hits.

7 Optimal Control with non-linear dynamics and costs

As before, we assume an unknown function

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) + \text{noise}(t)$$

We are given a cost function, which is known by the controller:

$$c(t) = \mathit{Cost}(\mathbf{x}(t), \mathbf{u}(t))$$

The task is to minimize one of the following expressions:

$$\sum_{t=0}^{\infty} c(t) \quad \text{or} \quad \sum_{t=0}^{t_{\max}} c(t) \quad \text{or} \quad \sum_{t=0}^{\infty} \gamma^t c(t) \text{ where } 0 < \gamma < 1 \quad \text{or} \quad \lim_{n \to \infty} \frac{1}{n} \sum_{t=0}^{n} c(t)$$

The attractive aspect of these formulations are their extreme general purposeness. All of the previous control formulations are special cases of at least one of these.

The delayed rewards nature of these tasks means that actions we choose at time t do not only affect the quality of the immediate reward but also affect the next, and all subsequent states, and in so doing affect the future rewards attainable. This leads to immense computational difficulties in the general case.

A large literature on such learning control problems has sprung up in recent years, with the general name of *reinforcement learning*. Overviews may be found in [Sutton, 1984, Barto *et al.*, 1989, Watkins, 1989, Barto *et al.*, 1994, Moore and Atkeson, 1993]. In this paper we will restrict discussion to the applications of memory-based learning to these problems.

Again, we proceed by learning an empirical forward model. In this case, though, the controller design is computationally much more expensive, although conceptually easy. A general-purpose solution can be obtained by discretizing state-space into a multidimensional array of small cells, and performing a dynamic programming method [Bellman, 1957, Bertsekas and Tsitsiklis, 1989] such as value iteration or policy iteration to produce two things:

1. A value function, mapping cells onto the minimum possible sum of future costs if one starts in that cell.

2. A policy, mapping cells onto the optimal action to take in that cell.

Value iteration can be used in conjunction with learning a world model. It is, however, extremely computationally expensive. For a fixed quantization level, the cost is exponential in the number of state variables. The most computationally intensive version would repeat value iteration after every update of the memory base:

Algorithm: Mem-Based-RL

```
1. Observe the current state $\mathbf{x}(t)$ and choose action $\mathbf{u} = \boldsymbol{\pi}(\mathbf{x})$
2. Perform action, and observe next state $\mathbf{x}(t+1)$
3. Add $(\mathbf{x}(t), \mathbf{u}) \rightarrow \mathbf{x}(t+1)$ to the memory base.
4. Recompute the optimal value function and policy using value
   iteration and the new model.
```

Because of the enormous expense of value iteration, the dynamic programming would normally be performed only at the end of each trial, or as an incremental parallel process in the manner of [Sutton, 1990b, Moore and Atkeson, 1993, Peng and Williams, 1993].

Experiment design

This algorithm does not explicitly explore. If the learned model contains serious errors, a part of state space which wrongly looks poor will never be visited by the real system, and so the model will never be updated. On the other hand, we do not want the system to explore every part of state space explicitly—the supposed advantage of function approximation is the ability to generalize parts of the model without explicitly performing an action. To resolve this dilemma, a number of useful exploration heuristics can be used, all based on the idea that it is worth exploring only where there is little confidence in the empirical model [Sutton, 1990b, Kaelbling, 1990, Moore and Atkeson, 1993, Cohn *et al.*, 1995].

Example: Container Filling

The problem involves filling containers with variable numbers of non-identical products. The product characteristics also vary with time, but can be sensed. Depending on the task, various constraints are placed on the container-filling procedure regarding the total weight of a batch and minimum weight of individuals.

Conventional practice is that controls for such equipment are chosen by human operators, but this choice is not easy as it is dependent on the current product characteristics and the current task constraints. The dependency is often difficult to model and highly non-linear. For example, for one given control parameters setting the amount of wastage varies approximately according to product mean weight and standard deviation according to the relationship in Figure 10. Furthermore, product characteristics drift randomly during the shifts.

In our experiments, the state of the system was four dimensional (plus time), and container-filling was optimized over batches of several thousand containers. Locally weighted regression learned a dynamic model, and value iteration was performed over a 160,000-cell decomposition of state space. The experimental result were deemed a success, and learned a considerably better controller quickly.

Experimental details are not available, and so instead we illustrate this form of learning by means of a very simple simulated example. Figure 11

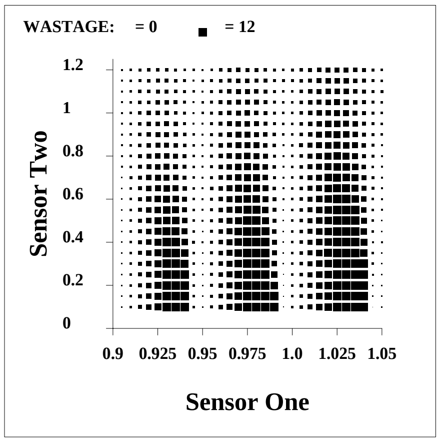


Figure 10: Wastage as a function of product characteristics: all other features held constant.

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 insufficient to accelerate up the slope. If the goal is at the hill-top, a strategy that proceeded by greedily choosing actions to thrust towards the goal would get stuck.

This is made clearer in Figure 12, 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 13, 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.

We performed two experiments.

- **Experiment 1: Conventional Discretization.** This used the conventional reinforcement learning strategy of discretizing state space into a grid of 60×60 cells. The reinforcement learning algorithm was chosen to be the as efficient as possible (in terms of data needed for convergence) given that we were working with a fixed discretization. All transitions between cells experienced by the system were remembered in a discrete state transition model. A learning algorithm

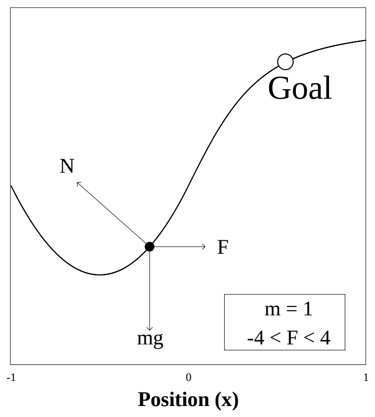


Figure 11: A frictionless puck acted on by gravity and a horizontal thruster. The puck must get to the goal as quickly as possible. There are bounds on the maximum thrust.

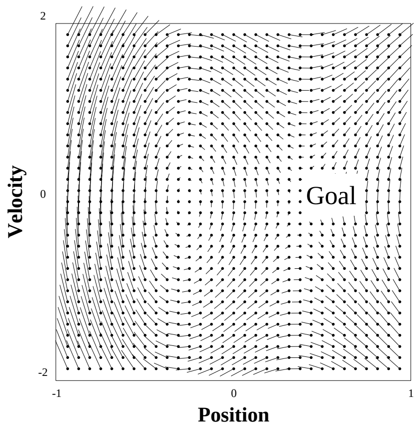


Figure 12: The state transition function for a puck that constantly thrusts right with maximum thrust.

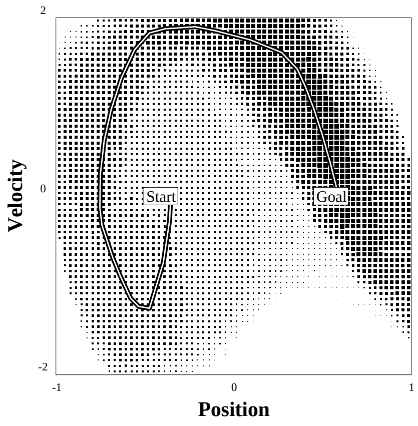
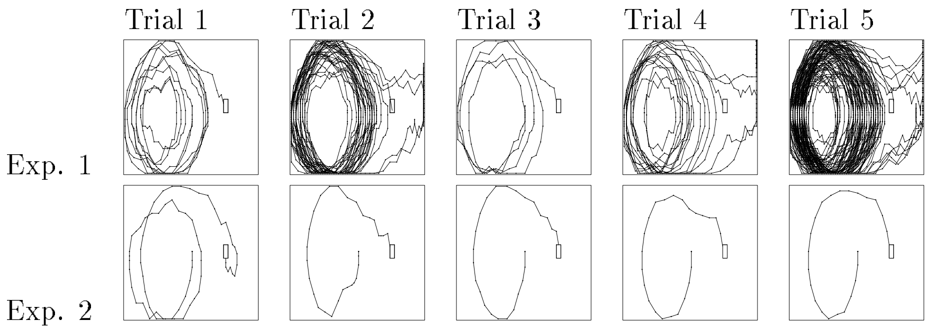


Figure 13: The minimum-time path from start to goal for the puck on the hill. The optimal value function is shown by the background dots. The shorter the time to goal, the larger the black dot. Notice the discontinuity at the escape velocity.

similar to Dyna [Sutton, 1990b] was used with full value iteration carried out on the discrete model every time-step. Exploration was achieved by assuming any unvisited state had a future cost of zero. The action, which is one-dimensional, was discretized to five levels: $\{-4N, -2N, 0N, 2N, 4N\}$.

- **Experiment 2: LWR-based Model.** The second experiment was the same as the first, except that transitions between cells were filled in by predictions from a locally weighted regression forward model $\mathbf{x}(t+1) = \hat{\mathbf{f}}(\mathbf{x}(t), \mathbf{u}(t))$. Thus, unlike experiment 1, many discrete transitions which had not been physically experienced were stored in the transition table by generalization from the real experiences.

The experimental domain is a simple one, but its empirical behavior demonstrates an important point. Generalizing the forward model in combination with value iteration can dramatically reduce the amount of real-world data needed. The graphs of the first five trajectories of the two experiments are shown in the following table.



In total, the number of real-world transitions needed by the experiments before permanent convergence to a near optimal path (within 3% of optimal) was:

Experiment 1 (discrete model)	28,024 steps to converge
Experiment 2 (model generalized with LWR)	291 steps to converge

The computational costs of this kind of control are considerable. Although it is not necessary to gather data from every part of the state space when generalization occurs with a model, the simple form of value iteration requires a multidimensional discretization for computing the value function. The following question is the subject of considerable ongoing research: How can we reduce the cost of value iteration when a model has been learned? (e.g. [Moore, 1991b, Mahadevan, 1992, Atkeson, 1994]).

8 The consequences of the memory-based approach

The memory-based approach leads to fairly different kinds of autonomous control than have been previously studied in the literature. This is primarily because the resulting algorithms explicitly perform empirical modeling as well as designing their controllers. Since this approach is different, it is worth discussing the strengths and weaknesses.

- **Flexibility, adaptive bias and adaptive resolution of local methods.**

  Cross-validation and local-kernel-optimization approaches can accommodate (i) functions with critical, fine, levels of detail, (ii) functions with considerable noise to smooth away, (iii) functions with differing input relevance and scaling. Some critical parts of the state space can be represented at very fine detail while other sparser, less important regions are represented merely with their general trends.

- **Automatic, empirical, local Taylor expansions.** Locally weighted regression returns a local linear map. It performs the job of an engineer who is trying to empirically linearize the system around a region of interest. It is not difficult for neural net representations to provide a local linear map too, but other approximators such as straightforward nearest neighbor or CMAC [Albus, 1981, Miller, 1989] are less reliable in their estimation of local gradients—in the case of CMAC this is because predictions are obtained from groups of locally flat tiles.

- **Automatic, confidence estimations.** Locally weighted regression can also be modified to return a confidence interval along with its prediction. This can be done heuristically, with the local density of the data providing an uncertainty estimate [Moore, 1991a] or by making sensible statistical assumptions [Schaal and Atkeson, 1994b, Cohn *et al.*, 1995]. In either case, this has been shown empirically to dramatically reduce the amount of exploration needed when the uncertainty estimates guide the experiment design. The cost of estimating uncertainty with memory-based methods is small. Neural networks can also be adapted to return confidence intervals [MacKay, 1992, Pomerleau, 1994], but approximations are required, and the computational cost is larger. Worse, any parametric model (such as global polynomial regression or a neural net) that predicts confidence statistically is making the assumption that the true world can be modeled by at least one set of its parameters. If this assumption is violated the confidence intervals lose any useful interpretation. We are unaware of any methods for obtaining confidence intervals from a CMAC trained online.

- **Adding new data to a memory-based model is cheap.** In contrast, users of neural networks for control must make a choice. Either they only use each piece of new data once, or else they must remember all data and repeatedly pass all data through the net each time a new experience arrives. Neither solution is satisfactory.

- **One-shot learning.** We do not make the same mistake twice. When we make an error there is no small incremental adjustment to compensate. Instead we can just switch to the action most likely to succeed given the single new datapoint. In some cases, the convergence to optimal performance can be superlinear (reducing error by a factor K requires only $O(\log\log K)$ steps.

- **Non-linear, yet no danger of local minima in function approximation.** Locally weighted regression can fit a wide range of complex non-linear functions, and finds the best fit analytically, without requiring any gradient descent. There are no dangers of the model learner becoming stuck in a local optimum.

- **No interference.** We don't care about what we task we are currently learning or if data distribution changes. In contrast anything trained incrementally with a delta rule, or linear adaption rule eventually forgets old experience and concentrates all representational power on current experience.

- **Costs increase with data.** Memory and prediction costs increase with the amount of data. Memory costs are clearly linear, and are not generally a problem. Even if the system makes 10 observations a second and operates for three months it only generates 10^9 items of data, which can easily be stored on fast access storage.

  Computational costs are more serious. Although computers and signal processors are powerful enough to cope with large memory-bases, there will always be a limit to the memory-base size that can be accommodated in real-time. One approach is to simply throw away some data, perhaps selected according to predictive usefulness or age. A more attractive solution is to structure the data so that very close approximations to the output predicted by locally weighted regression can be obtained without explicitly visiting every point in the database. There are a surprisingly large number of algorithms available for doing this, mostly based on kd-trees [Preparata and Shamos, 1985, Omohundro, 1987, Moore, 1990, Grosse, 1989, Quinlan, 1993, Omohundro, 1991, Deng and Moore, 1995].

- **Delta-rule increments are difficult.** It is very easy to add and remove datapoints from the model. But some other learning control schemes use a delta-rule update step when learning control, for example [Miller *et al.*, 1987, Miller, 1989, Jordan and Rumelhart, 1992]. Standard memory-based methods cannot be updated in this way.

9 Discussion

There are two questions that can be asked. How useful is it to learn models in order to learn control? And what is the benefit of using memory-based methods to learn those models? This paper has examined and empirically evaluated both questions—throughout the bulk of the paper, the emphasis was on the ways that forward and inverse learned models can be used. The experiments were all performed with memory-based models. The previous section discussed in more detail the pros and cons of memory-based function approximators as the specific choice of model learner.

Perhaps the most important issue is autonomy. Why would we be interested in learning control as an engineering application unless the goal is to reduce the amount of human programming, decision-making and expertise needed to control a process or robot? In this survey the autonomy has been strengthened in the following ways:

- A mathematical model of the system to be controlled does not need to be derived. The model is obtained empirically.
- The empirical modeling ("learning from data") is itself highly automatic. The wide class of functions that can be learned by memory-based methods, and the wide variety of self tuning methods (cross-validation, local kernel optimization etc.) reduce the need for human tweaking.
- The controller is also automatically designed. Depending on the class of control task, this can happen in a number of different ways.
- The training phase ("experiment design") is also automated. All the algorithms we have seen decide themselves where they need data.

There is little doubt that these victories for autonomy can be converted into general purpose packages for the benefit of robotics and process control. But it should also be understood that we are still a considerable distance from full autonomy. Someone has to decide what the state and action variables are for a problem, how the task should be specified, and what class of a control task it is. The engineering of real-time systems, sensors and actuators are still required. A human must take responsibility for safety and supervision of the system. Thus, at this stage, if we are given a problem, the relative effectiveness of learning control, measured as the proportion of human effort eliminated, is heavily dependent on problem-specific issues.

References

[Aboaf *et al.*, 1989] E. W. Aboaf, S. M. Drucker, and C. G. Atkeson. Task-Level Robot Learning: Juggling a Tennis Ball More Accurately. In *IEEE International Conference on Robotics and Automation*, 1989.

[Albus, 1981] J. S. Albus. *Brains, Behaviour and Robotics.* BYTE Books, McGraw-Hill, 1981.

[Atkeson *et al.*, 1995] C. G. Atkeson, A. W. Moore, and S. Schaal. Lazy learning with Locally weighted regression: A survey of algorithms, statistics, experiment design and parameter selection. Submitted to AI Review Special Issue on Lazy Learning, 1995.

[Atkeson, 1989] C. G. Atkeson. Using Local Models to Control Movement. In *Proceedings of Neural Information Processing Systems Conference*, November 1989.

[Atkeson, 1994] C. G. Atkeson. Using Local Trajectory Optimizers to Speed up Global Optimization in Dynamic Programming. In S. J. Hanson, J. D Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, April 1994.

[Barto *et al.*, 1989] A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and Sequential Decision Making. COINS Technical Report 89-95, University of Massachusetts at Amherst, September 1989.

[Barto *et al.*, 1994] A. G. Barto, S. J. Bradtke, and S. P. Singh. Real-time Learning and Control using Asynchronous Dynamic Programming. *AI Journal, to appear (also published as UMass Amherst Technical Report 91-57 in 1991)*, 1994.

[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.

[Cohn *et al.*, 1995] D. A. Cohn, Z. Ghahramani, and M. I. Jordan. Active learning with statistical models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7.* MIT Press, 1995.

[Conte and De Boor, 1980] S. D. Conte and C. De Boor. *Elementary Numerical Analysis*. McGraw Hill, 1980.

[Deng and Moore, 1995] K. Deng and A. W. Moore. Multiresolution Instance-based Learning. In *To appear in proceddings of IJCAI-95*. Morgan Kaufmann, 1995.

[Grosse, 1989] E. Grosse. LOESS: Multivariate Smoothing by Moving Least Squares. In L. L. Schumaker C. K. Chul and J. D. Ward, editors, *Approximation Theory VI*. Academic Press, 1989.

[Jordan and Rumelhart, 1992] M. I. Jordan and D. E. Rumelhart. Forward Models: Supervised Learning with a Distal Teacher. *Cognitive Science*, In press, 1992.

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

[MacKay, 1992] D. J. C. MacKay. Bayesian Model Comparison and Backprop Nets. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*. Morgan Kaufmann, April 1992.

[Mahadevan, 1992] S. Mahadevan. Enhancing Transfer in Reinforcement Learning by Building Stochastic Models of Robot Actions. In *Machine Learning: Proceedings of the Ninth International Workshop*. Morgan Kaufmann, June 1992.

[Mel, 1989] B. W. Mel. MURPHY: A Connectionist Approach to Vision-Based Robot Motion Planning. Technical Report CCSR-89-17A, University of Illinois at Urbana-Champaign, June 1989.

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

[Miller *et al.*, 1987] W. T. Miller, F. H. Glanz, and L. G. Kraft. Application of a General Learning Algorithm to the Control of Robotic Manipulators. *International Journal of Robotics Research*, 6(2), 1987.

[Miller, 1989] W. T. Miller. Real-Time Application of Neural Networks for Sensor-Based Control of Robots with Vision. *IEEE Trans. on systems, Man and Cybernetics*, 19(4):825–831, July 1989.

[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 *et al.*, 1992] A. W. Moore, D. J. Hill, and M. P. Johnson. An Empirical Investigation of Brute Force to choose Features, Smoothers and Function Approximators. In S. Hanson, S. Judd, and T. Petsche, editors, *Computational Learning Theory and Natural Learning Systems, Volume 3*. MIT Press, 1992.

[Moore, 1990] A. W. Moore. Acquisition of Dynamic Control Knowledge for a Robotic Manipulator. In *Proceedings of the 7th International Conference on Machine Learning*. Morgan Kaufmann, June 1990.

[Moore, 1991a] A. W. Moore. Knowledge of Knowledge and Intelligent Experimentation for Learning Control. In *Proceedings of the 1991 Seattle International Joint Conference on Neural Networks*, July 1991.

[Moore, 1991b] 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 Kaufmann, June 1991.

[Moore, 1992] A. W. Moore. Fast, Robust Adaptive Control by Learning only Forward Models. In J. E. Moody, S. J. Hanson, and R. P. Lippman, editors, *Advances in Neural Information Processing Systems 4*. Morgan Kaufmann, April 1992.

[Omohundro, 1987] S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987.

[Omohundro, 1991] S. M. Omohundro. Bumptrees for Efficient Function, Constraint, and Classification Learning. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann, 1991.

[Ortega and Rheinboldt, 1970] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, 1970.

[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.

[Pomerleau, 1994] D. Pomerleau. Reliability estimation for neural network based autonomous driving. *Robotics and Autonomous Systems*, 12, 1994.

[Preparata and Shamos, 1985] F. P. Preparata and M. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[Quinlan, 1993] J. R. Quinlan. Combining Instance-Based and Model-Based Learning. In *Machine Learning: Proceedings of the Tenth International Conference*, 1993.

[Sage and White, 1977] A. P. Sage and C. C. White. *Optimum Systems Control.* Prentice Hall, 1977.

[Schaal and Atkeson, 1994a] S. Schaal and C. Atkeson. Robot Juggling: An Implementation of Memory-based Learning. *Control Systems Magazine*, 14, 1994.

[Schaal and Atkeson, 1994b] S. Schaal and C. G. Atkeson. Assessing the Quality of Local Linear Models. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, April 1994.

[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 Kaufmann, 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 Kaufmann, June 1990.

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