Robot Skill Learning Through Intelligent Experimentation

by

Jeff G. Schneider

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Christopher M. Brown

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1995

Acknowledgments

My first thanks are to my advisor, Chris Brown. He gave me the freedom and encouragement to pursue my craziest ideas, while always being there to bail me out when things went bad. His knowledge of the literature always kept me focused. From the time of my first paper on, he exhibited an amazing ability to make up the time I squandered during writing by quickly providing thorough comments. I am also indebted to my other committee members: Dana Ballard for his inspirational talks and positive attitude, Randal Nelson for his mathematical rigor, and Roger Gans for teaching me about dynamic modeling.

I have benefited greatly from technical discussions with many of my fellow students. I would especially like to thank Lambert Wixson, Virginia de Sa, Polly Pook, Andrew McCallum, and Jonas Karlsson. I would also like to thank Bob Wisniewski for his tennis game. Many of the ideas in this thesis were come upon while trying to figure out how to return his forehands and serves. I owe thanks to Jason Chu for his programming work on the simulator used to perform the experiments in this dissertation and to George Ferguson for supporting much of the other software in use at Rochester. I would like to thank everyone on the staff at Rochester for keeping the place running smoothly, but specifically Tim Becker for his work in the robot lab and his willingness to make my problems his problems.

Finally, there are the people most important to me. I would like to thank my parents for the values they instilled in me and for being there when I needed them. It is unlikely they will ever realize how much strength I got from their silent support. I would like to thank Ginny for standing by me with patience and love. Her smile got me through the stress of completing this dissertation.

This research was supported by DARPA research grant number MDA972-92-J-1012, NSF research grant number IRI-9306454, U.S. Air Force research grant number AFOSR-91-0288, and ONR research grant number N00014-93-1-0221.

Abstract

In robot skill learning the robot must obtain data for training by executing expensive practice trials and recording their results. The thesis is that the high cost of acquiring training data is the limiting factor in the performance of skill learners. Since the data is obtained from practice trials, it is important that the system make intelligent choices about what actions to attempt while practicing. In this dissertation we present several algorithms for intelligent experimentation in skill learning.

In open-loop skills the execution goal is presented and the controller must then choose all the control signals for the duration of the task. Learning is a high-dimensional search problem. The system must associate a sequence of actions with each commandable goal. We propose an algorithm that selects practice actions most likely to improve performance by making use of information gained on previous trials. On the problem of learning to throw a ball using a robot with a flexible link, the algorithm takes only 100 trials to find a "whipping" motion for long throws.

Most closed loop learners improve their performance by gradient descent on a cost function. The main drawback of this method is convergence to non-optimal local minima. We introduce the concept of *cooperation* as a means of escaping these local minima. We assume the existence of several *coaches* that each improve some aspect of the controller's performance. Switching training between coaches can help the controller avoid locally minimal solutions. On the task of curve tracing with an inverted pendulum the cooperative algorithm learns to track faster than with a traditional method.

In an integrated system with scarce sensor resources it is preferable to perform tasks without sensing. We observe that closed loop learning can function as an efficient search technique for open-loop control. Our system starts with closed loop learning. As it improves its ability to control the plant, it replaces sensor information with estimates. The result is a controller that tracks long segments of a reference curve open loop.

Table of Contents

List of Tables

List of Figures

1 Introduction

How does a basketball player fly through the air, avoid the bodies of several defenders, and launch the ball on a path that passes through the hoop? How does a hockey player maintain balance while also avoiding defenders and steering the puck toward the net? How does a baseball pitcher capable of running only 20 mph find the motion to propel a baseball at speeds up to 100 mph? Each of these athletes performs a highly skilled action. Each spends hundreds of hours in training to learn and hone their skill. This thesis work is motivated by a desire to find out how machines might also learn to perform highly skilled tasks such as these. We do not aspire to finding out the true mechanisms used by highly skilled human athletes, but do look to what is known about human skilled performance for inspiration.

In order to approximate athletic skills reasonably, the robot motor skills we consider have several important characteristics. The first is that they are nonlinear. Humans must deal with nonlinearities in their limb kinematics and dynamics and in their neural control system [Jeannerod, 1988]. Furthermore, the control of linear systems is already well understood and can often be done without learning. Second, the skills are parameterized. Possible parameters include speed of execution and distance traveled by the robot or a projectile. A baseball pitcher throws at a range of speeds and at a variety of targets given by the catcher and a basketball player shoots the ball through the hoop from all over the court. The acquisition of single, unparameterized tasks can often be done with existing techniques from learning and optimal control. Finally, we prefer to consider skills that are difficult to model. Control of systems that can be accurately modeled is often done best by using the model to construct a controller without learning.

1.1 Robot skill learning

This thesis and other existing work are summarized and compared in Figs. 1.1 and 1.2. We believe that learning to perform a motor skill open loop is preferable to doing it closed loop because the performance (speed at which it can track a reference curve for example) of a closed loop system is limited by its sensors. The sensors may hinder performance with their inaccuracy, limited bandwidth, and excessive delay. Therefore, the discussion of motor skills begins at the box on the left of Fig. 1.1. Chapter 3

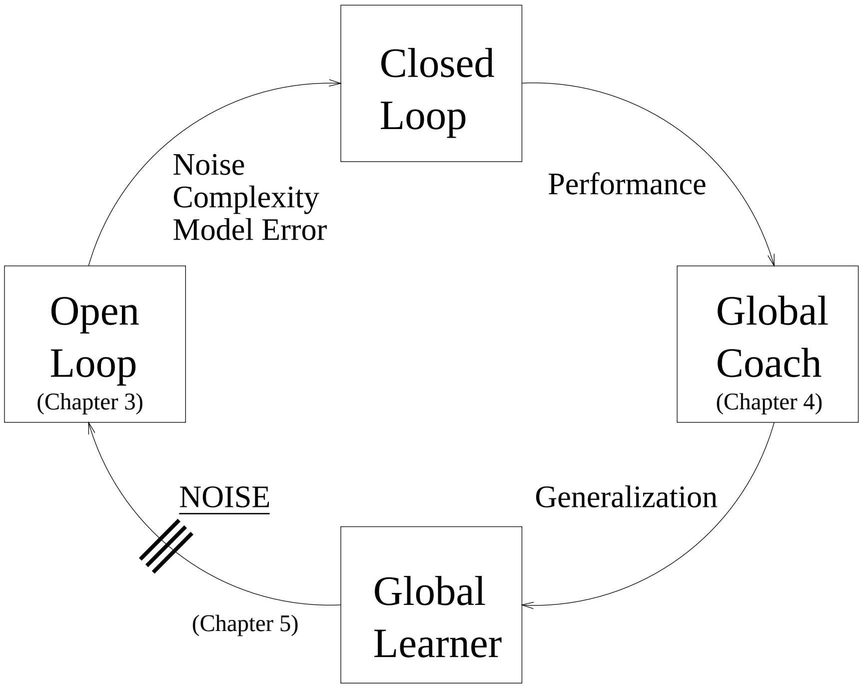

Figure 1.1: Robot skill learning big picture

discusses the direct approach to open-loop skill learning and gives an algorithm for doing so efficiently.

Unfortunately, it is not possible to learn all skills open-loop. Noise in the plant and modeling error can make it necessary to add sensors and operate closed loop for many skills. The complexity of the task may also call for the use of sensing. If it is impossible to search the space of all action sequences (even with the methods proposed in Chapter 3) it may be necessary to reduce the control problem to choosing single actions by using sensor information at each time step. That brings us to the other three boxes in Fig. 1.1 which all represent closed loop learning.

Fig. 1.2 shows the relationship between closed loop control, learning control, and skill learning. One of the oldest forms of control is linear control. A control signal is computed at each time step as a linear combination of terms based on past and current errors between desired and actual states. Linear control is very well understood and versatile, but its performance is sometimes limited by the fact that it only considers the current desired state.

Linear controllers attempt to get as close as possible to the current desired state without considering whether doing so will put them in a less desirable state with respect to future goals. Optimal control attempts to find the control that obtains the best performance in terms of the current and all future desired states. It may trade off how near it gets to the current goal state in exchange for doing much better on future goals. Optimal control uses knowledge of the entire task to obtain the best overall performance.

The performance of fixed controllers is limited by the accuracy of the plant model

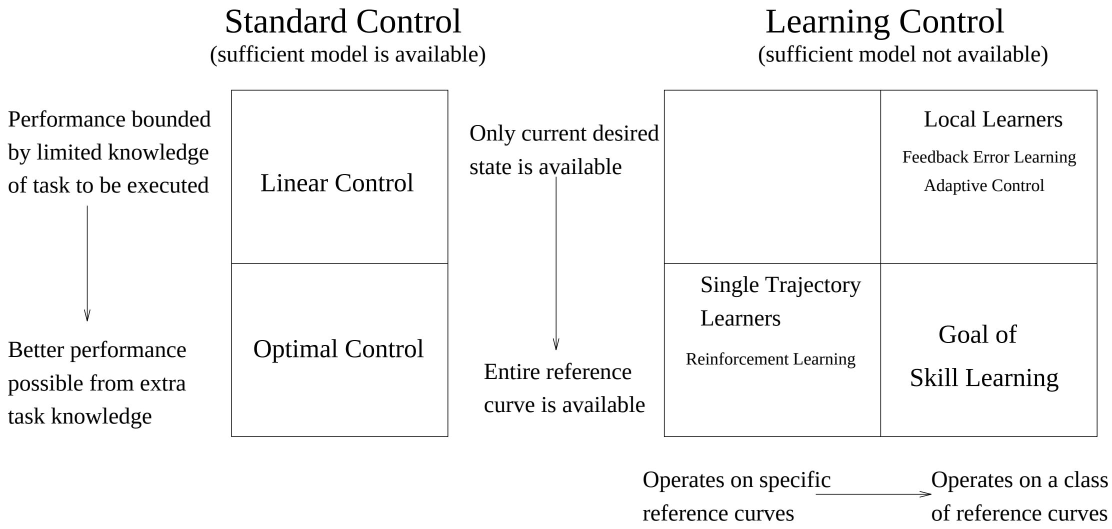

Figure 1.2: The relationship of skill learning to closed loop control

they use. If it is impossible to get a sufficient plant model, it becomes necessary to employ learning techniques to build a good controller. Local learners are the learning analog to linear control (represented as the upper right quadrant in Fig. 1.2 and the "closed loop" box in Fig. 1.1). The information they use to compute modifications to their control strategy is based only on the current state of the plant, control signal, and error. These learners work well on tasks where improving the performance at the current time does not negatively impact performance later in the task.

Just as optimal control is used to outperform linear control when the entire reference curve can be made available, learning controllers can also perform better when the whole task is considered in the computation of modifications to the control strategy. We refer to this as a "global coach" as in Fig. 1.1. The learning method is like the coach of a figure skater that says, "bend your knees more as you approach the jump." Bending the knees may seem to sacrifice a current goal (maintaining balance), but ultimately realizes a greater future goal (such as jumping higher). Chapter 4 discusses "global coaches" and shows the performance benefits of using them.

There is a trap in the use of global coaches though. Often the learning controller still only considers the current information about the reference curve when making its control decision. Suppose the learner enters the same state during trials on two separate tasks. On the first, the coach says "lean farther right" because he knows a right turn is approaching, but on the second he says "lean farther left" because he knows a left turn is approaching. The learner, however, sees only the current state which is identical in both cases. Even though the coach knew exactly what to do in each case, it becomes impossible for the learner to store the information. The result can be a "single trajectory learner" as shown in the lower left quadrant of Fig. 1.2. If the same task is repeated over and over, there is no conflict in what the action should be for each state and desired state combination. Unfortunately, this results in the learning of only one specific task and the inability to perform others. Reinforcement learning is one example of this kind of learning. The dynamic programming used in reinforcement learning is closely related to some of the methods used to solve optimal control problems. Note that the upper lefthand box in Fig. 1.2 is left blank. Most learners that acquire only one reference curve ultimately encode it into their internal representation and thus fall in the same category as the single trajectory learners. What would be preferable is a learning controller that can make use of the optimal control properties of a global coach while retaining the ability to perform on any curve as the local learners do. This is the lower right quadrant of Fig. 1.2.

The bottom box in Fig. 1.1 represents the last step of a solution to the skill learning problem. A switch from standard "closed loop" learning to the use of a "global coach" was made to improve performance. Next a move to a "global learner" gives generalization across a class of reference curves. A "global learner" is one that also uses information about the future reference curve when available to determine its control signals. This allows it to differentiate how to optimize performance when identical current states and goals are reached, but future goals are drastically different.

Note that a learning controller with full knowledge of the task to be performed and the optimal controls to use for it may no longer need to use its sensors after its initial state has been determined. Closed loop learning control, including a global coach and global learner, can be used just as a sophisticated method of searching a very complex action space for an open-loop learning controller. Of the reasons that drove us to closed loop control, the only one left is noise. Only the amount of noise present in the plant limits the ability to return to open-loop control. Even if it is not possible to do open-loop control, the end result of learning may be a significantly reduced reliance on sensors. This reduced reliance has the benefit of allowing better performance, but also pays off in integrated intelligent systems that must deal with contention for sensor resources. As the system learns to perform a particular skill with less sensor data, sensors are freed up for other tasks. The use of learning to reduce use of sensor resources is discussed in Chapter 5.

1.2 Contributions

This thesis addresses the following problems:

- Open-loop skills usually come with high-dimensional action vectors. This is primarily because there is at least one dimension for each time step in the task. Learning open loop skills involves searching the space of action vectors. Since searching involves execution on the robot and the number of trials that can be executed in a reasonable amount of time is small, open loop learning is not possible with standard naive search tactics.

- Most closed loop learning schemes define a cost metric and compute derivatives of cost with respect to the controller parameters. They learn by adjusting the controller parameters with gradient descent. Gradient descent, and thus most closed loop learners, fails if the cost function has local minima at unacceptable performance levels.

- Integrated systems may attempt to perform several tasks with limited sensing resources. In such systems it is desirable to execute each task with as little sensory information as possible. Unfortunately, many tasks can only be learned with high-bandwidth, low-delay sensory feedback. The problem for integrated systems, then, is how to convert a skill that is learned closed loop into one that runs open loop, or at least with reduced need for sensing resources.

Solutions to these problems are proposed in Chapters 3, 4, and 5, respectively:

- We develop a search algorithm that combines interpolation, extrapolation, and random search with a consideration of existing memory entries. The efficiency and goal-directedness of the algorithm allow it to learn skills that can not be acquired with existing search techniques.

- We propose cooperation between learning controllers to avoid getting stuck in local minima. Several learning controllers that each tend to improve a different aspect of performance are used. A selection algorithm considers the current system performance and activates the learning controller most likely to improve it. By switching between several learning controllers, this algorithm avoids getting stuck in the local minima of any single one.

- We use closed loop learning as an efficient search technique for open-loop execution. As the system learns, it replaces sensor information with estimates in order to reduce the amount of sensor resources it requires. The end result is a controller that executes long sequences of actions without sensing, thus freeing up those resources for other controllers in the same integrated system.

1.3 Recurring themes

The three main contributions of this thesis work can be taken almost independently, but there are several recurring themes throughout. We summarize them here so the reader can understand the motivation for this work and watch for them to appear in each chapter.

The most important theme is summed up in the title of this dissertation. The standard paradigm in learning often takes the form:

1. A set of training data is compiled.

2. The learning algorithm processes the data to maximize its performance against some metric.

The implicit assumption is that the training data is supplied for free and that it contains all the information necessary to perform well against the metric. In robot skill learning the training data is not free [1]. It is obtained at high cost by operating a physical system (the robot) and observing the result. Since it is so expensive to obtain training data, the amount of data is limited and the selection of what data to acquire is crucial to the final performance of the learner. This is where the term "intelligent experimentation" comes from. We believe the limiting factor in robot skill learning is a system's ability to choose its experiments in order to provide the best possible training data for learning.

Often a skill learner has multiple goals. Common goals include performing as fast as possible, or as accurately as possible, or with as little effort as possible. These goals usually interact with each other. Therefore, learning to achieve one goal often functions only as a step toward achieving some other goal. For example, learning to perform a task with low effort often leads to the ability to perform it faster. The problem of achieving multiple interacting goals is approached explicitly through the cooperative mechanisms of Chapter 4, and implicitly throughout the rest of the work.

Learning has often found itself at odds with engineering. The ultimate success for a learning researcher seems to be an algorithm that learns with "no a priori knowledge." At the same time, an engineer has only conquered a problem when a complete model has been obtained. In practice, it is nearly impossible to build a learning algorithm without assumptions. Similarly, there are numerous difficult control problems whose solution can not wait for a full scientific understanding. Therefore we approach the skill acquisition problem as one of combining the information we can obtain from modeling with the information we can obtain through learning.

1.4 Roadmap for this dissertation

Chapter 2 provides the background information necessary to understand the rest of this document and begins with a glossary of terms relevant to robot motor skill learning. All of the learning algorithms presented here are based on the use of function approximators. The workings of these are explained with emphasis on the ones actually used for our experiments. It is easiest to discuss robot motor skills in terms of control theory, so a few relevant concepts are explained in Chapter 2. These include linear control, optimal control, and state estimation. Finally, related work is discussed at the end. Any or all of Chapter 2 may be skipped by the reader already familiar with the material.

The three major contributions of this work are presented in chapters 3, 4, and 5 and may be read almost independently of each other if the relationships described in Figs. 1.1 and 1.2 are well understood. Chapter 3 presents an algorithm for doing open-loop learning from the perspective of it being a search problem in a high-dimensional space. Chapter 4 discusses the concept of a global coach for closed loop learning, but observes

[1] At the end of Chapter 3 we discuss a mechanism that substitutes simulations for some real executions to obtain cheaper training data, but the expensive training data assumption still holds unless all real executions can be eliminated.

that even learning with a perfect gradient coach may suffer from convergence to a non-optimal local performance minimum. To remedy the problem, it proposes the concept of "cooperation" whereby several coaches work together to help a controller learn a skill without getting stuck in local minima. Chapter 5 assumes the closed loop learning problem has been solved for some skill and turns its attention to the freeing of sensor resources. It presents an algorithm that chooses not to use its sensors whenever it deems their information unnecessary. As learning progresses the algorithm leaves the sensors off for longer periods of time, thereby making them and their associated computational resources available for other uses.

Chapter 6 is the summary and conclusions for this dissertation. It also discusses what we believe are the most important remaining problems in robot motor skill learning.

2 Robot Skill Learning Background

Robot skill learning draws from at least four fields: human motor physiology, robotics, control theory, and learning theory. Many researchers in artificial intelligence get inspiration from biological systems and ours comes from human athletic performance. No formal background in this area is required, however, to understand this dissertation. We assume the reader has a basic understanding of robotics concepts. A good introductory book is [Craig, 1986]. Relevant background information from the two remaining fields is the subject of this chapter.

Most authors use the terms in skill learning loosely so we start this chapter with some definitions. The learning algorithms presented in this dissertation depend on the use of continuous function approximators. This chapter reviews the basics of function approximation with special emphasis on the methods used in later chapters. Following that we discuss concepts from control theory including PID control, optimal control, and state estimation. Optimal control is important since we typically evaluate the success of our control learners using the metrics of optimal control. State estimation is important background for Chapter 5 where we discuss reducing dependence on sensors. Finally, we discuss related robot learning work.

2.1 Definitions

- **skill** or **motor skill** - The ability to produce an entire class of desired results beginning from some set of initial conditions. Examples include being able to throw an object (or class of objects) to hit a set of target locations or being able to trace a class of curves with some device.

- **task** - A single instance of some skill. For example, throwing a ball to hit a specific target, or tracing one particular curve.

- **execution** or **trial** - One attempt at a particular task.

- **open-loop** - A skill or task for which no sensory information is available during its execution. Sensors may, however, provide initial state information beforehand, and may provide final state information afterward for success evaluation.

- **closed loop** - A skill or task for which some plant state information is provided during execution.
- **plant** - The part of the system that must be controlled to produce some desired result (in this work "plant" often means "robot").
- **state** - A set of parameters describing the current condition of the plant (for example, the current angular positions and velocities of a robot's joints).
- **control signal**, **action**, or **command** - a signal input to the plant that will cause some change in the state and output of the plant.
- **controller** - The part of the system that chooses control signals in an attempt to produce a set of desired results at the plant's output (for example, a controller might choose torques for a robot's joints).
- **skill learner** - A controller that is modified during or after executions in an attempt to improve its performance.
- **intelligent control** - Applied generically to advanced forms of control and often used to mean learning and control.
- **cost**, **penalty**, or **performance** - A function that converts the results of an execution, or set of executions into a single scalar value indicating how well the controller is doing. Lower values are usually better for cost or penalty and higher values are usually better for performance.
- **system** - Generically used to describe the hardware and software of the plant and controller together.
- **local learner** - A closed loop learning controller that bases its learning entirely on the changes needed to reduce the cost at the current time step without consideration of future cost.
- **single trajectory learner** - A learning controller that must be trained separately for each reference curve it tracks.
- **noise** - Any distortion to a process that can not be predicted deterministically by existing models. In control it shows up as alterations in state variables that can not be accounted for by the model or values from sensors that do not equal the underlying quantity being measured. In function approximation it is a difference between the values indicated by the sample data and the real values of the function at those data points. [1]

[1]There are two schools of thought on noise. One (Fisher viewpoint) assumes that the system being controlled is deterministic and noise results only from inaccurate modeling and sensing. The other (Bayes viewpoint) assumes the system is inherently stochastic and must be modeled probabilistically. We prefer the intuition behind the former veiwpoint, however this choice does not affect the algorithms proposed in this work.

2.2 Function approximators

Function approximators (FAs for short) are tools commonly used in learning and classification. An FA assumes the existence of a function, $y = f(x_1, \ldots, x_n)$, that it will attempt to mimic. The FA has a set of parameters, w_j, that determine the function it computes. We indicate the function actually computed by the FA with a caret as in:

$$\hat{y} = \hat{f}(\vec{w}, \vec{x}) \tag{2.1}$$

Normally, the contribution of the parameters, $\vec{w}$, to the FA is assumed implicitly and they are not listed explicitly as arguments of $\hat{f}$. We refer to $\vec{x}$ as the input or query and $\hat{y}$ as the output of the function approximator. If the output comes from a small finite set of values, the FA is called a classifier. This dissertation does not consider any classification problems.

The FA trains on a set of data points: $y_i \leftarrow (x_{1,i}, \ldots, x_{n,i})$. The data is assumed to be derived from the real function, f, being approximated. It may represent the function exactly, or may be distorted by a number of processes referred to as noise. The function approximator has a learning rule that uses the data to set the value of its parameters, $\vec{w}$, with the goal of making $\hat{f}$ similar to f in relevant parts of the function input space.

FAs may be trained off-line (batch), or on-line (iteratively). Off-line training occurs when all the data is presented at the beginning and used to make a one-time setting of the FA's parameters. On-line training occurs when the data is presented over a length of time and the FA always maintains the best approximation it can given the data it has seen so far. We prefer on-line methods for skill learning problems since we want to execute the learning algorithms throughout the life of a robot and have it continue to improve its performance as its experience increases.

A good FA has several important properties. We have alluded to the main one already. It should "learn" in the sense that its computation, $\hat{f}$, should approach f as it receives more training data. In order to learn, the FA should have the ability to "generalize." This means that it should be able to provide reasonable mappings even for inputs, $\vec{x}$, that it has not received training data for. Most FAs generalize by a kind of interpolation between data points. Unfortunately, many do not explicitly address the problem of producing reasonable values outside the region of data provided (extrapolation) and inadvertently do a poor job of it. An FA may need to overcome significant noise to do its learning. As mentioned above, the data may not exactly represent the desired function and the FA should cope with that. Two sources of distortion to data are noise in the operation of a system, and in the data collection process. Finally, the FA may be used to approximate a time-varying function, $y = f(t, x_1, \ldots, x_n)$, and should change its approximation as f changes through time. In summary, a good function approximator should:

1. Improve its approximation as it gets more training data.

2. Generalize.

3. Deal with noisy data appropriately.
4. Adjust its approximation to handle time-varying functions.

Function approximators all have assumptions about the shape of the function they are approximating. Since a finite amount of data can not determine a function, any hypothesized function is chosen by appealing to some assumptions. The most prominent assumption is that functions are "smooth." Smoothness can have various definitions, but common ones are that the function be continuous, or have continuous derivatives up to some level, or have bounded derivatives. Smoothness assumptions leave function approximators with two competing goals. They want to reproduce the training data as well as possible, but must deal with noisy training data by preferring smoothness over exact reproduction.

This dissertation is *not* concerned with the design of good function approximators. We believe the most important issue in skill learning is the selection of actions to obtain training data, not what the FA does with the training data. Our work on skill learning assumes the existence of FAs that have the desirable properties listed above. Any FA with these properties can be used with any of the algorithms presented in the later chapters. In this section we describe some common function approximators, paying extra attention to those used in the experiments described later.

2.2.1 Memory-based methods

Memory-based function approximation methods (sometimes known as table lookup) are among the easiest to understand, so we start with a description of them. When training data is presented, a memory-based FA simply stores the data in its memory. Later, when a request is made to produce a value at some point, it uses the stored values to compute a reasonable answer by one of a variety of methods. Here we summarize the basic possibilities.

Nearest neighbor

When a nearest neighbor FA receives a query, it searches its memory for the data element nearest the input point. It then returns the output associated with that data element. The thin line in Fig. 2.1 shows the form of $\hat{f}$ when it is computed by this method. The diamonds are the data points provided to the function approximator, and thus stored in its memory. In the one-dimensional case, the FA output is a series of steps as shown by the dashed line. Note that this method is simple, but produces a somewhat undesirable looking result in that it is discontinuous and thus violates the usual assumption that most of the functions we approximate are reasonably smooth.

Nearest neighbor can be improved slightly with the addition of interpolation between data points. In the one-dimensional case, the nearest neighbor to the left and right of the input is found and the output is determined with simple interpolation. This results in a slightly nicer function (it is continuous) as shown by the thick line in Fig. 2.1. We

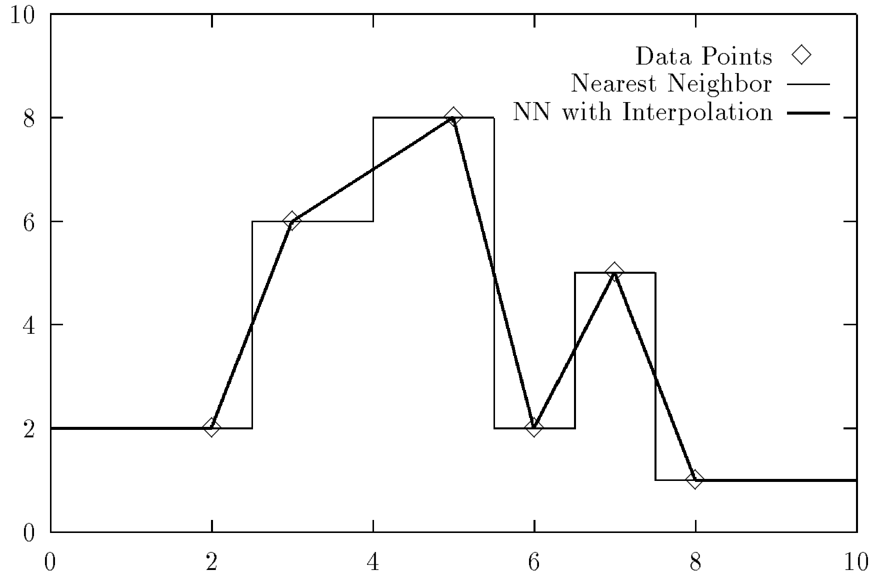

Figure 2.1: Function approximation with nearest neighbor and interpolation

assume the mapping for the ball throwing system in Chapter 3 is continuous, but has little noise. Therefore we want the FA to give a continuous result, but do not want it to sacrifice accuracy for smoothness in an attempt to handle noise. Based on those assumptions we choose nearest neighbor with interpolation for its FA.

Local modeling

Interpolation makes the approximated function continuous, but its derivative is still discontinuous. Furthermore, it is fairly sensitive to noise. One erroneous data point will ruin the approximated function for all values between it and the neighboring good data points. One way to solve both of these problems is local modeling. When a query arrives, a set of points near the input is retrieved and fit to a model assumed to hold locally for the function being approximated. Normally the points nearest the input point contribute more to the model fitting than those farther away. The simplest type of weighted local modeling is a weighted average (which assumes a constant local model). Given n data pairs (x_i,y_i), the value of the approximated function is defined by:

$$f(x) = \frac{\sum_{i=0}^{n-1} \frac{y_i}{1+(x-x_i)^2}}{\sum_{i=0}^{n-1} \frac{1}{1+(x-x_i)^2}} \tag{2.2}$$

Typically, only data points near x are considered. The curve in Fig. 2.2 shows the form of a weighted average function approximation.

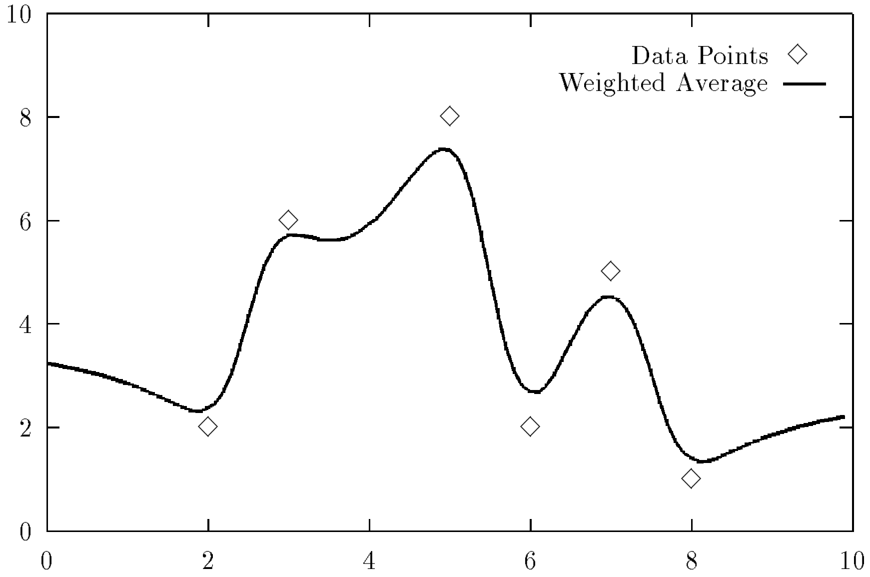

Figure 2.2: Function approximation with local modeling

The weighted average FA is smoother than the nearest neighbor FAs in that it has continuous first derivatives, but it still has some undesirable characteristics. Note that the function does not pass very near the given data points. Also, the values it provides outside the range of the given data points may not seem realistic. The data could be interpreted as having a downward trend, but the globally higher data points pull the estimated values up instead. These problems can be addressed by changing the local model. Common alternatives are linear or quadratic models. For example, use of a local linear model would call for fitting a line to the n points in the neighborhood with a weight on their contributions. For a query point, x, a weighted regression is done by finding the values m^* and b^* that minimize the cost function:

$$g(m,b) = \sum_{i=0}^{n-1} \frac{(y_i - mx_i + b)^2}{1 + (x - x_i)^2} \tag{2.3}$$

The FA outputs: $\hat{y} = m^*x + b^*$. The two examples presented here use the weighting function $\frac{1}{1+(x-x_i)^2}$, but any function that weights the nearer neighbors more will work. The shape of the weighting function also affects the form of the resulting approximation. Examples of robot learning with local linear models can be found in [Atkeson, 1991; Schaal and Atkeson, 1993a; Schaal and Atkeson, 1994].

Retrieval efficiency and other limitations

Memory-based function approximation is attractive because it follows a policy of "least commitment." When it receives training data it makes no decision about how to use it for future queries, and just stores the data instead. This policy allows any technique to be applied later when a query to the FA arrives. Unfortunately, this policy also calls for significant memory resources and can require significant computation time to return an answer. A large amount of computation can be spent just finding the set of points nearest the query point.

A common solution to the problem is to store data in a structure called a *kd-tree*. A kd-tree is similar to the simple binary trees used to store one-dimensional data. It is modified to store and retrieve multi-dimensional data and does so in logarithmic (as a function of the number of nodes) time just as a simple binary tree does. Each node of a binary tree splits the data into two parts according to the one-dimensional key it contains. In a kd-tree the split is made along a different dimension of the data at each level of the tree. Usually, the dimensions are rotated through cyclically down the tree. Kd-trees are important for efficient manipulation of the multi-dimensional data usually found in skill learning. Moore gives an excellent analysis of the use of kd-trees for learning in his thesis [Moore, 1990]. Even with kd-trees, the amount of memory required to implement a memory-based method may be prohibitive. The rest of the function approximators listed here do not increase their memory usage with each new presentation training data.

A final drawback to simple memory-based FAs is that they treat all data equally. This can be a problem when the FA is used to approximate a time-varying function. For these functions, training data received a long time ago may not carry much information about the current state of the function, and may actually worsen the resulting approximation. Time-varying functions are common in large systems where several FAs work together. For example, the algorithm in Chapter 5 has one FA modeling a plant and a second one approximating how well the first does its job. As the first FA learns more, the function being approximated by the second changes. As the system learns and changes itself to approximate a time-varying function, old data becomes irrelevant and should be discarded. We say the FA must "forget" the old data as it becomes irrelevant. A good study of these "forgetting" algorithms can be found in [Salganicoff, 1992; Salganicoff, 1993]. The rest of the function approximators discussed below solve the forgetting problem implicitly because they use a finite amount of memory. Since training data is processed and discarded, the contribution of any particular data point is diminished over time.

2.2.2 Radial basis functions

Radial basis function FAs (RBFs) are a form of memory-based learning. Given a set of data points, $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$, the function approximator is defined as:

$$\hat{f}(\vec{x}) = \sum_{i=1}^{n} w_i G(\| \vec{x} - \vec{x}_i \|) \tag{2.4}$$

The method is called radial basis functions because G is a function of the distance between the query point, $\vec{x}$, and the training data points, $\vec{x}_i$. The functions chosen for G (gaussians are common, for example) are often the same as those used for weighting data points in local modeling memory-based FAs. The learning rule for RBFs solves a set of equations to choose the w_i such that the training data are perfectly reproduced. With an appropriate G, a smooth interpolation between the training data points results.

Since RBFs are just another memory-based method, there have been numerous modifications to alleviate the problems they share with other memory-based methods. The vectors $\vec{x}_i$ are often called the radial basis function "centers." One method of relieving the memory consumption problem is to fix the number of centers chosen and not increase them with additional training data presentations. It is then necessary to find learning rules to choose the position of the centers, $\vec{x}_i$, and the weights at the centers, w_i. With sufficient amounts of data the usual system of equations becomes overconstrained. Usually, a form of gradient descent is employed to set the parameter values. A cost function is defined in terms of the error in reproducing the data and the parameters are adjusted to reduce this error. Examples of RBF variants can be found in [Poggio and Girosi, 1989; Poggio and Girosi, 1990; Moody and Darken, 1989].

In Chapter 4 we employ an FA based on RBFs. We preset the position of the centers in a grid pattern and do not change them during learning. The radial function G is hyperbolic:

$$G(\vec{x}, \vec{x}_i) = \frac{1}{c_1 + (\| \vec{x} - \vec{x}_i \| / c_2)^2} \tag{2.5}$$

We modify the computation of eq. 2.4 as in [Moody and Darken, 1989]:

$$\hat{f}(\vec{x}) = \frac{\sum_{i=1}^{n} w_i G(\vec{x}, \vec{x}_i)}{\sum_{i=1}^{n} G(\vec{x}, \vec{x}_i)} \tag{2.6}$$

Eq. 2.6 computes an average of the w_i weighted by G. This is convenient because the w_i can be interpreted as the actual value of the function near $\vec{x}_i$. We further speed up the computation by considering only the centers near the query point. The process of finding the near centers is simplified by the fact that the centers are arranged in a grid. These modifications simplify the learning algorithm too. When training data is presented we simply move the w_i whose centers, $\vec{x}_i$, are in the vicinity of the training data, toward the training output value. The resulting FA bears as much resemblance to the model-based methods described earlier and the Kohonen maps described below as it does to plain RBF function approximation.

2.2.3 Neural Networks

The most common function approximators are probably those defined broadly as "neural nets." A good introductory text is [Hertz *et al.*, 1991]. Here we review two types of these: back-propagation networks [Rumelhart *et al.*, 1986; Werbos, 1974] and Kohonen maps [Kohonen, 1988; Ritter *et al.*, 1992]. They are referred to as neural nets because aspects

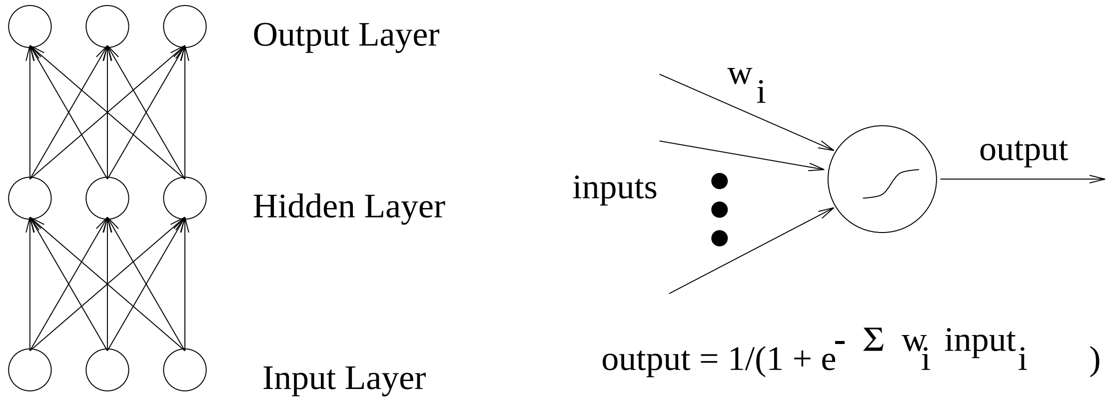

Figure 2.3: Back-propagation net and individual neuron

of their architecture are inspired by biological neural models. The models are formed by connecting numerous elementary computing units. Most authors call them “neurons” although the computation they perform does not necessarily bear much resemblance to the computation performed by real biological neurons.

Back-propagation nets

Fig. 2.3 shows the basic architecture of a back-propagation neural network. It is based on the model of an individual neuron. The neuron receives a set of inputs and computes the dot product of its weights, w_i, and the inputs. The result is modulated by an activation function. The most common activation function is the sigmoid: $f(x) = 1/(1+e^{-x})$. It is important that the activation function be nonlinear so extra representational power can be gained by stacking the neurons into multiple layers and that it be differentiable everywhere so gradient descent may be used to train the network. The result of the activation function is sent as the output of the neuron.

The neurons are usually composed into layers. The input layer has its values fixed by the input signal. The number of layers and units within each layer may be modified to alter the properties of the network. The adjustable parameters for this function approximator are the weights on the connections between neurons. Their modification is done by gradient descent on the error at the output. For each presentation of training input, the net is run forward to compute its output. Then the desired output is compared with the actual and the derivative of the error with respect to each weight is computed. Finally, all the weights are changed in the negative direction of the gradient.

In practice, back-propagation networks often suffer from slow convergence. The slow convergence is partially because of another undesirable feature. Each weight in the model tends to affect the output of the FA throughout the entire input space. The result is a type of interference during learning. If the system goes through a phase of learning where all the samples are from one region of the input space, the network will sacrifice its approximation in the other regions in order to best fit the recent data. The difference between global function approximator parameters (such as with back-propagation) and

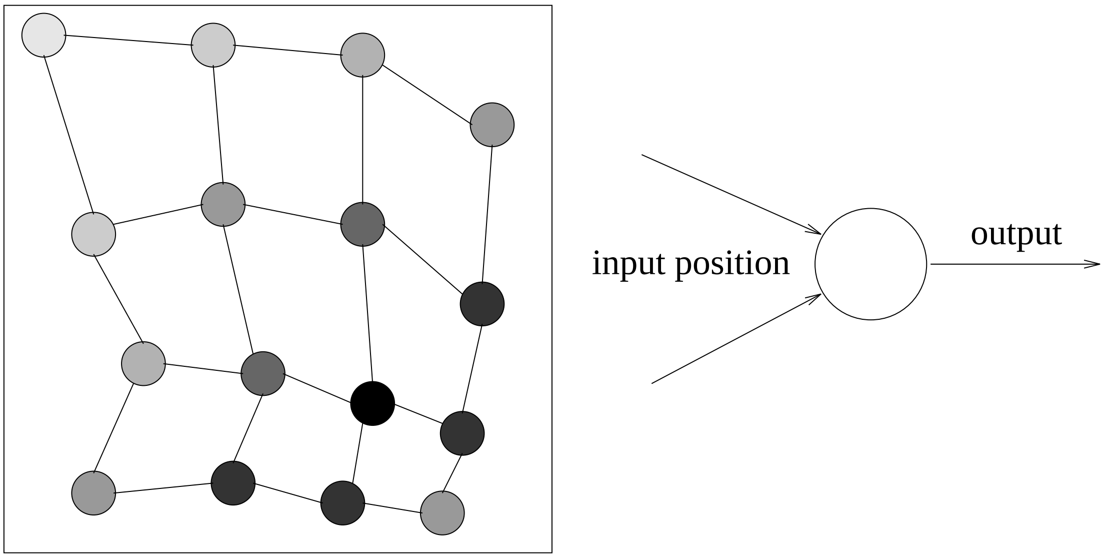

Figure 2.4: 2-d Kohonen map and individual neuron. Input is coded as position in the map and output is coded as gray level

local parameters (such as with nearest neighbor) is analogous to the difference between doing experimental data regression with polynomials vs doing it with splines.

Kohonen maps with outputs

Here we describe a Kohonen map that has been augmented with trainable outputs as proposed in [Ritter *et al.*, 1992]. In the description to follow, the equations concerning $\vec{w}^{out}$ are from Ritter, while the rest are from the original work by Kohonen. Fig. 2.4 shows the architecture. The individual neuron does not perform a computation the way back-propagation neurons do. It simply stores an input position and an output value. The neurons are arranged in a regular grid topology. The connecting lines indicate the neighbor relations for the grid. The position of a neuron within the box represents the weights of its two input lines. Gray level shows the output values of each neuron.

There are three values that define a neuron in a Kohonen map. The position of the neuron in the input space is denoted by $\vec{w}^{in}$. Its position in the grid topology (e.g. second neuron in the third row) is denoted by $\vec{r}$. Finally, its output value is denoted by $\vec{w}^{out}$. The output value for this FA is given just as in eq. 2.6:

$$\hat{f}(\vec{x}) = \frac{\sum_{s \in A} \vec{w}_s^{out} G(\vec{x}, \vec{w}_s^{in})}{\sum_{s \in A} G(\vec{x}, \vec{w}_s^{in})} \tag{2.7}$$

where A represents a set of nodes whose $\vec{w}^{in}$ values are within some radius of the query input, $\vec{x}$. The function G is commonly gaussian.

The grid topology is fixed so the $\vec{r}$ are fixed. The modification of the weights on the presentation of training data is as follows. For training data $(\vec{x} \rightarrow \vec{y})$ find the node, s' whose input weights are nearest the data point:

$$\| \vec{w}_{s'}^{in} - \vec{x} \| \leq \| \vec{w}_{s}^{in} - \vec{x} \| \quad \forall s \tag{2.8}$$

Update the input weights according to:

$$\Delta \vec{w}_{s}^{in} = \alpha_1 G_1(\vec{r}_s, \vec{r}_{s'})(\vec{x} - \vec{w}_{s}^{in}) \tag{2.9}$$

and the outputs according to:

$$\Delta \vec{w}_{s}^{out} = \alpha_2 G_2(\vec{r}_s, \vec{r}_{s'})(\vec{y} - \vec{w}_{s}^{out}) \tag{2.10}$$

The learning algorithm for the input weights is important because it drags more nodes to areas of the input space where there have been more training data samples. In terms of RBFs, this is Kohonen's solution to the placement of centers. It is useful because it increases the resolution of the mapping where it is needed most. It can be problematic, however, if an important area of the input space receives no training data for a period of time. This results in a shift of nodes away from that area and is a type of interference as seen with back-propagation networks. The learning rule for the output weights is similar to the RBFs described above.

Other neural networks

Competitive learning [Rumelhart and Zipser, 1986; Grossberg, 1976] is an unsupervised learning technique that is strongly related to Kohonen networks. Unsupervised learning means there are only input data points, but no associated desired outputs. The learning algorithm just groups all the input data into clusters. Since we are interested in computing continuous mappings rather than just clustering, competitive learning is not particularly useful in skill learning. Because it is unsupervised, it generally does not have a trainable output as Kohonen maps do. The positioning of nodes in the input space is done exactly as Kohonen maps, except that there is no topological grid and the arguments to G are the positions in the input space rather than the grid. Usually, G is one at s' and zero elsewhere. The addition of an output value to competitive learning would allow it to become a supervised method and another possible FA for robot skill learning.

CMACs [Albus, 1981] are another function approximator similar to neural networks. Values are stored in a table whose dimensionality matches that of the input space. Outputs are determined by computing the sum of several table entries near the input vector. In its plain form, it has the drawback of a fixed generalization width. This is because the output is computed from values in the same size neighborhood regardless of how much experience has been accumulated in the table. There have been modifications to standard CMACs that use more sophisticated methods to combine table entries [Lane *et al.*, 1992].

2.3 Concepts from control theory

It is impossible to do justice to the vast literature on control theory. Instead, we use this section to highlight a few specific areas and point out their relevance to the work presented in later chapters.

2.3.1 Open-loop control

Open-loop control is among the simplest methods of control. A block diagram for an open-loop controller is shown at the top of Fig. 2.5. The controller looks only at the desired output of the plant and produces a control signal based on it. The main drawback of open loop control is that the controller must use an exact model of the plant. Any difference between the plant model used by the controller and the real plant will create errors in the output of the plant. A difference may arise due to noise in the plant, a change in the operating characteristics of the plant, or the inability of the designer to produce an accurate plant model.

2.3.2 Feedback control

The bottom of Fig. 2.5 is a block diagram of a simple feedback, or closed loop, control system. The plant is a system whose output we would like to control. The desired output of the plant is input as the reference signal. The controller uses that information and information from its sensors to compute a signal to send to the plant. The plant is represented by a set of state variables and equations.

$$\dot{\vec{x}} = f(\vec{x}, \vec{u}, t) \tag{2.11}$$

$$y = g(\vec{x}, t) \tag{2.12}$$

where $\vec{x}$ represents the current state of the plant, $\vec{u}$ is the control signal, y is the output, and t is time. If the functions f and g are linear in $\vec{x}$ and $\vec{u}$, the plant is linear. If f and g are not functions of t, the plant is time-invariant. Control of linear, time-invariant systems is the best understood and can be read about in numerous text books [Dorf, 1986; Ogata, 1987] . The systems discussed in this dissertation are generally considered to be nonlinear time-invariant.

PID control

There are many types of linear control, but we will only describe a common method called PID (Proportional Integral Derivative) control. It is often used as the starting point for learning control systems including feedback error learning [Kawato *et al.*, 1987] which is discussed in detail in Chapter 4. It is defined as follows. Compute the error as the difference between the desired and actual output of the plant, $e = y_d - y$. Produce control u with:

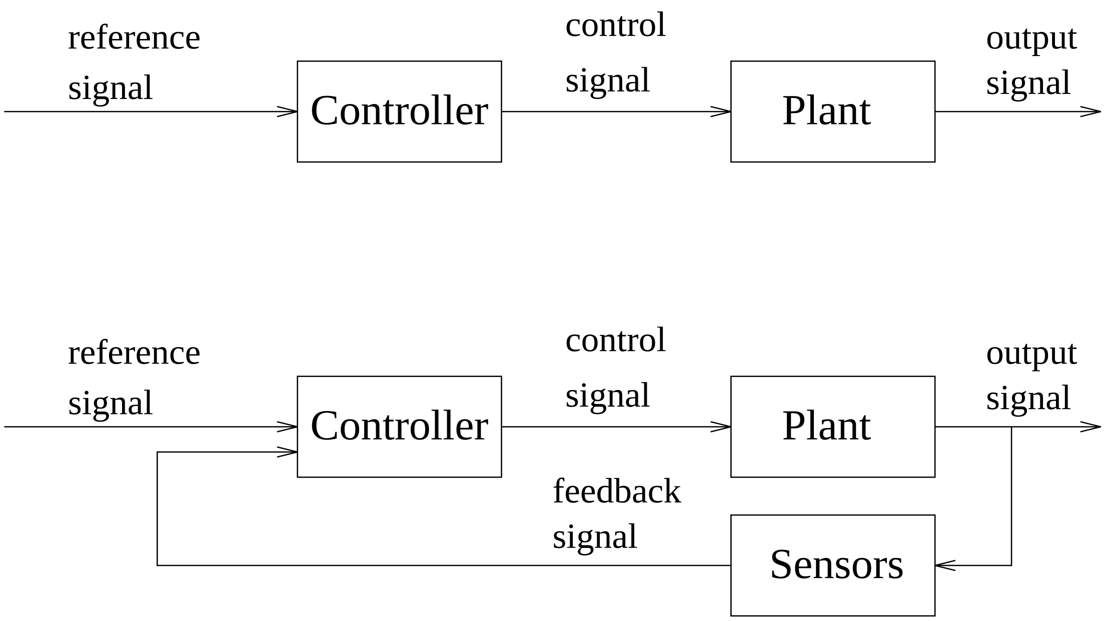

Figure 2.5: Block diagram for open-loop (top) and feedback (bottom) control

$$u = k_p e + k_i \int e dt + k_d \frac{de}{dt} \tag{2.13}$$

where the constants k_p, k_i, and k_d are set according to the characteristics of the plant and the desired system performance. The simplest version of this controller is when k_i and k_d are zero. Then the controller just sends a control proportional to the current difference between the desired and the actual plant output. With only this first term, it is possible that the system will reach a steady state before the error becomes zero. In that case, the integral term is added. If the error remains nonzero, the integral of the error will continue to increase and cause an increase in the control signal until the error is reduced to zero. The controller may also cause increasing oscillations between positive and negative error. The function of the derivative term is to eliminate them. As the error begins to decrease, the control will be reduced in advance of it actually reaching zero. Similarly, if the error is increasing the control will be further increased by the derivative term. The setting of the three PID constants for linear time-invariant plants can be read about in any linear control text [Dorf, 1986; Ogata, 1987].

Drawbacks of feedback control

Feedback control uses its sensors to reduce tracking error, but its sensors can also be a liability. Sensors may produce significant measurement errors or even fail outright. More importantly, feedback control fundamentally limits the speed at which a reference curve can be tracked.

Real plants and real sensing systems exhibit delay. Delay is the time it takes for the results of a control action to be seen by the controller through its sensors. There may be delay in the plant and the sensors. Unfortunately, delay can cause a system to become unstable. For a qualitative understanding of the problem, consider a controller trying to position a robot arm. Its sensors tell it that the arm should be raised so it exerts a torque in the upward direction. Because of delay, the sensors continue to say the arm should go higher even after it has reached the desired height and the controller continues to push higher. By the time the sensors report that the arm has reached its goal, it is actually well past. Eventually they will show that the arm needs to be lowered and the controller will exert a downward torque. This cycle can continue resulting in oscillations in the position of the arm. If the gain is low enough (i.e. the arm moves slowly enough toward its goal) the oscillations will eventually die out. If the gain is too high, they will increase, making the system unstable. The greater the delay is, the slower the controller must move the arm in order for it to remain stable. A mathematical analysis of the stability of feedback systems can be found in any linear control text [Dorf, 1986; Ogata, 1987].

Negative feedback systems are those like the PID controller that subtract the actual from the desired state to get error and compute a control based only on the error. For these systems the gain (defined as the output signal divided by the input signal) is fundamentally limited. Consider the two block diagrams in Fig. 2.5. Lump the controller and the plant together and call its transfer function G. The transfer function of the sensors is H. The gain for the open-loop system is G, and the gain of the closed loop system is $\frac{G}{1+GH}$. If H is one (it commonly is) then the gain can never exceed one. However, the gain in the open-loop case can be increased infinitely. This gain limitation is another reason the performance of feedback systems is restricted. The gain determines how fast the output can change as a function of changes in input.

Note that the gain of a negative feedback system can exceed one if H is set below one. Since H is the gain of the sensors, reducing it effectively reduces the contribution of the sensors to they system's control. In the limit as H goes to zero, the sensors are not used, and the closed loop gain equation equals the open loop gain equation.

One of the goals of a skill learner is to perform its task as quickly as possible. As we have seen, though, the use of sensors puts a hard limit on how fast the task can be performed without loss of stability. It is for this reason that we believe the ultimate goal of a skill learner is to perform open loop. We present an algorithm for direct open-loop learning in Chapter 3 and an algorithm to switch smoothly between closed loop learning and open-loop execution in Chapter 5.

2.3.3 Optimal Control

The performance of linear controllers is often evaluated by the way they respond to step changes in a reference signal. Typical quantities of interest are the rise time (time it takes to get within a certain percent of the desired value), settle time (time it takes for oscillations to die down to a certain magnitude), overshoot (amount it goes past the desired value), and steady state error (difference between actual and desired output once oscillations have died out). There are design methods to build linear controllers with specified levels of performance under these criteria. The resulting controllers work acceptably for simple systems, but may not produce desirable performance for complex, nonlinear systems. Optimal control seeks to solve this problem by requiring an explicit cost metric to be minimized. The result is a controller that chooses a control signal at each step to minimize the cost incurred for the rest of the trial. There are numerous good text books on the topic ([Kirk, 1970; Lewis, 1986], for example). A good entry point into the literature on the use of optimal control in intelligent control is [White and Jordan, 1992].

An optimal control problem is specified by the state equations in eq. 2.11 and a cost metric of the form:

$$J = h(\vec{x}(t_f), t_f) + \int_{t_0}^{t_f} g(\vec{x}(t), \vec{u}(t), t)dt \tag{2.14}$$

where $\vec{x}(t)$ is the value of the state variables at time t, and t_0 and t_f are the beginning and ending times. The first term only applies to systems that have a specific ending time. The cost function is typically constructed to penalize tracking error, control effort, and time required to complete the task. We show the standard optimal control cost function here because it will be used as a measure of performance throughout this dissertation.

An optimal controller is one that, at each time step, looks at the current state, the current and future desired states, and the cost function and chooses a control that will minimize the cost incurred from the current time step on. Note that this may entail sacrificing accuracy or effort at the current step to reduce the costs paid at future steps. This tradeoff will be important when considering the inverted pendulum problem in Chapter 4.

There are several ways to approach the solution to optimal control problems. A classical method is to use the calculus of variations [Leitmann, 1981]. This method is not discussed further as we assume our problems are too difficult or do not come with the necessary prior information to be solved by it. It is also possible to employ numerical techniques. An interesting method known as *back-propagation-in-time* [Pearlmutter, 1989; Pineda, 1989; Simard, 1991] uses the methods of back-propagation to implement a kind of numerical solution. Fig. 2.6 shows one implementation for this type of solution. The neurons are the same as the ones used in standard back-propagation. We assume that the mapping from current state and control to next state, which is computed in the rightmost two layers of the network, has already been learned. The goal is to train the weights of the leftmost two layers, which take a current state and a desired state and compute a control signal. Note that the important output of the system is the control signal that leaves the network at the bottom center. The controller is run on a reference curve and the errors are computed at the rightmost layer of the network by subtracting the values there from the desired values presented at the left. Then the back-propagation algorithm is used as before with some changes. The computation starts with the error at the last time step. The weights used to compute the next state are left unchanged and the error just propagates through them. When the error reaches

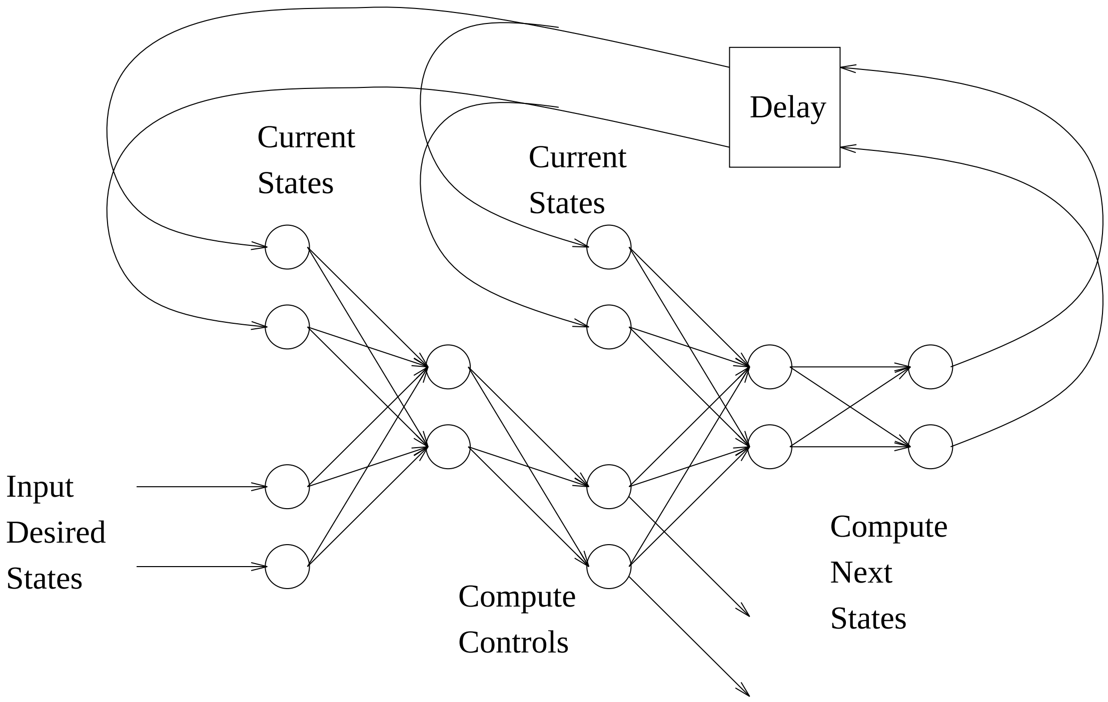

Figure 2.6: Optimal control via back-propagation-in-time

the weights used to compute the control, modifications are made. The error propagation continues backward through the loop with delay lines and is combined with the error on the previous time step at the righthand neurons. The error is propagated around the loop back through time until the beginning of the execution is reached. The equations for doing the back-propagation are nearly identical to the equations used for some gradient descent based numerical methods.

Another important method for solving optimal control problems is dynamic programming [Bellman, 1957; Bellman and Dreyfus, 1962]. To use it, the state and control spaces must be discretized. Then dynamic programming finds the sequence of (discrete) controls that minimizes the cost function. The significance of dynamic programming for optimal control is that it is at the heart of the learning paradigm called "reinforcement learning."[Sutton, 1984; Watkins, 1989] Reinforcement learning associates a reward (or cost) value with each state. It assumes that the transition function mapping states and actions onto states and reward is unknown a priori. Therefore, the usual dynamic programming computation can only be done for single steps as the system chooses an action and moves to a new state. In the limit that the agent tries every action from every state numerous times, the computation converges just as dynamic programming would [Watkins and Dayan, 1992]. Reinforcement learning is discussed further in the related work section since it is a potential, but often inefficient, method of solving skill learning problems.

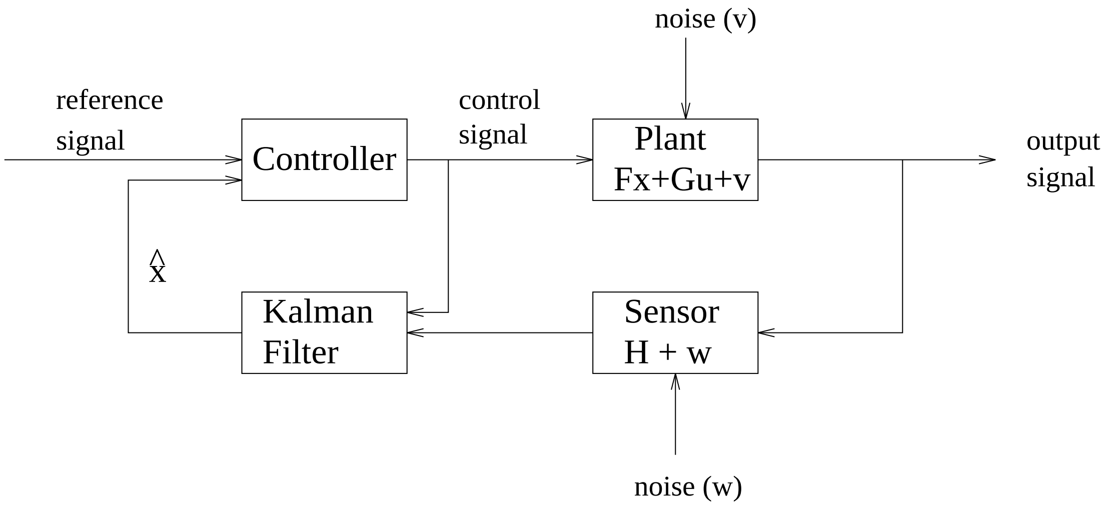

Figure 2.7: Kalman filtering for feedback control

2.3.4 State estimation with Kalman filters

Until now we have discussed control systems under the assumption that it is possible to sense the values of the plant state variables accurately. This is often not the case. There are numerous methods of estimating states from imperfect sensor information, but we choose to present Kalman filters [Bar-Shalom and Fortmann, 1988; Gelb, 1974] because of the interesting parallel that can be drawn between them and the algorithm presented in Chapter 5.

Fig. 2.7 shows how the Kalman filter is used for state estimation in a linear system with zero-mean, white, Gaussian noise. We assume the plant is specified by the equation:

$$x(k+1) = Fx(k) + Gu(k) + v(k) \tag{2.15}$$

where $v(k)$ is the noise entering at the top of the figure. We also don't have direct access to the state of the plant. Sensors measure information according to the equation:

$$z(k) = Hx(k) + w(k) \tag{2.16}$$

where $w(k)$ is the noise entering at the bottom of the figure. The Kalman filter makes use of the values $z(k+1), u(k), \hat{x}(k)$ and a model for F and G to produce an estimate of $\hat{x}(k+1)$. First, make an estimate of what the state should be in the absence of any sensor information and call it x^*.

$$x^*(k+1) = \tilde{F}\hat{x}(k) + \tilde{G}u(k) \tag{2.17}$$

Then compute the final estimate:

$$\hat{x}(k+1) = x^*(k+1) + W(k+1)(z(k+1) - Hx^*(k+1)) \tag{2.18}$$

Eq. 2.18 is a weighted average between the expected state and the observed state with W providing the weighting. As indicated by the $(k+1)$, W is iteratively estimated based on the history of the execution and the covariances of the noise, v and w, and includes the inverse of H.

The weighting is based on two goals which correspond to the two forms of noise entering the system. The plant noise, v, can be viewed as permanent damage to the estimation process since it alters the real state of the system. The filter must use its sensor information to counteract v since the estimated state can not account for it. The sensor noise, w, can be thought of as one-shot damage to each sensor reading. The filter must use its estimated state to cover up for the sensor fluctuations due to that noise. The fact that the noise is zero-mean is important to the computation of W and leads to the following intuitive description. If the errors between the predicted and actual sensor readings (second term of eq. 2.18) have zero-mean then assume the errors are due to sensor noise and weight the estimated term more heavily. If the errors do not have zero-mean, assume v has permanently altered the system and increase the weight on the sensor readings to bring the estimate back in line with reality.

Kalman filters use a weighted average between sensors and estimators to filter out noise. In Chapter 5 we make the assumption that noise is derived from inaccurate modeling of F during the learning process and use a switch (rather than a weighted average) between sensed and estimated state information. This allows the sensing resources to be freed up for other tasks whenever the estimated state is deemed sufficient.

2.4 Related skill learning work

The work directly related to our algorithms is discussed in the following chapters with the description of our work. Here, we discuss work in the general category of robot skill learning. Fig. 2.8 is an adaptation of Fig. 1.2 with references to existing work robot skill learning.

2.4.1 Reinforcement learning and dynamic programming

As noted earlier, reinforcement learning (RL) is one way to attack robot skill learning problems. However, it can be inefficient for many tasks. One problem is the way it approximates dynamic programming. It has no model of the state space and only updates a single step in its computation for each single step it makes. Since it has to take every step numerous times to ensure convergence [Watkins and Dayan, 1992], the number of executions can be prohibitive in real robotic systems. To date RL has been applied to *task* learning, but not to *skill* learning. The problem is that RL normally assumes a fixed goal (i.e. a single desired task). It is this fact that puts it in the lower left hand quadrant in 2.8. It would be possible to include a goal as part of the agent's sensory input and change the reward function appropriately, but this would increase the learning problem dimension and further emphasize the efficiency problems.

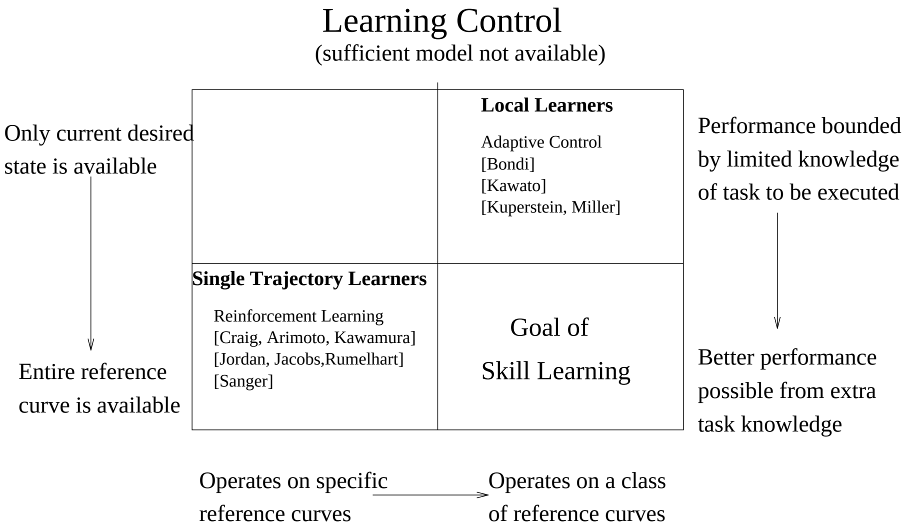

Figure 2.8: Related work in learning control

Several authors have looked at ways of reducing the number of real executions required in reinforcement learning. Sutton proposed DYNA which records the state transitions of individual steps and then uses them to perform simulated executions along with real ones [Sutton, 1990]. The real executions are most useful for obtaining the state transition information while the simulated ones cause more of the dynamic computation to be done. Moore and Atkeson further investigate methods of determining what parts of the dynamic programming computation are most important to compute between real trials [Moore and Atkeson, 1993]. Moore has also proposed using variable resolution in the state space to speed up RL [Moore, 1991; Moore, 1993]. McCallum uses a memory-based method to glean more information from the robot's executions [McCallum, 1994]. Atkeson [Atkeson, 1993] looks at ways of using local models to do local optimizations in dynamic programming which could be incorporated into a more efficient form of reinforcement learning.

Despite the work on improving efficiency, reinforcement learning does not seem appropriate for the problems we discuss later. The work in Chapters 3 and 5 deals with open-loop execution, for which RL is not well suited. The control problems of Chapter 4 are also difficult for RL because they are tracking problems rather than problems with a single goal state. RL normally assumes its cost is a function only of the state it is in. When the goal is to track a reference curve in time, cost becomes a function of state and time. One possibility is to encode the desired state into the RL agent's state space. Then the mapping from state to cost would be unique. Unfortunately, the effect would be that the agent moves along some dimensions of the state space (the actual current state) as a result of its actions, but get pushed around in the other dimensions (the desired state) by the external application of a desired state sequence. Modifying RL to handle curve tracking is an interesting problem, but we do not address it in this dissertation.

2.4.2 Model learning

Many learning control problems can be handled by the acquisition of a model for the robot. Kuperstein's INFANT [Kuperstein, 1988; Kuperstein and Rubenstein, 1989] is a simulated robot learning system that learns to reach for objects in its visual field. Since it is a kinematics learning problem, no intermediate sensing is required and this work fits into the Open Loop box in Fig. 1.1. The method is "inverse modeling." The plant's inputs are the joint angles for its arm. The output is the position of the hand in its camera image space. The mapping is learned by randomly positioning the arm and observing its hand position in the image. A function approximator is trained with the image position as its input and the joint angles as its output. Inverse modeling is attractive because the FA *is* a controller when the learning is done. No additional computation or training is required. If a forward model of the plant is learned, an additional step is necessary to train a controller for the system. Unfortunately, inverse modeling only works when the forward mapping is not redundant. If it is redundant, its inversion is not unique and the method employed by INFANT can not be used. Efficient learning of highly redundant mappings is discussed in Chapter 3.

Examples of forward model learning are [Miller *et al.*, 1987] and [Mel, 1990]. Miller is concerned with a robot that reaches for objects appearing in its visual field. Unlike Kuperstein, he learns a forward model first and then uses it to train a controller. Mel studies grasping for objects in the presence of obstacles. He uses the forward model to do a depth first search in the robots joint space to find a path to the object that does not cause the arm to hit any obstacles. These are also open-loop skill learning problems.

An important issue in model learning is how a partial plant model can be combined with learning to make the best use of existing knowledge. Raibert shows how dimensions in a function approximator can be traded off for extra parameters in a local model for the problem of learning robot dynamics [Raibert, 1977; Raibert, 1978]. For example, assume a robot arm needs to learn torque as a function of θ, $\dot{\theta}$, and $\ddot{\theta}$. One option is to use a function approximator with three dimensions. However, if the change in torque as a function of angular acceleration is known, an FA with only two dimensions (position and velocity) can be used. A local model mapping acceleration to torque (as in the memory-based methods discussed earlier) can be fit in different regions of the function approximator space and used in place of a third dimension. As the number of dimensions is reduced, the FA may learn quicker because it spans a smaller space, but it may also become more difficult to determine and fit the increasingly complex local model.

2.4.3 Gradient descent learning

Many robot control problems can be solved with gradient descent. Generally, a nominal controller initially operates the system and a gradient descent rule is applied to improve future executions. The learning of robot dynamics for single trajectories can be done with this method [Craig, 1984; Arimoto *et al.*, 1984; Kawamura *et al.*, 1988]. Once a nominal controller has been set up to follow a certain trajectory, the tracking is improved with a learning rule of the form:

$$u_{k+1,t} = u_{k,t} - \alpha \frac{\partial e_t}{\partial u_t} \tag{2.19}$$

where α is a learning rate, the t subscript indicates time and the gradient is determined by pre-specified model information. In a system with no noise, this scheme could result in open-loop execution as is discussed in Chapter 5. The main drawback is that each desired trajectory has to be learned individually. In some cases it is possible to decompose learned trajectories into parts that can be reused for new trajectories [Bondi *et al.*, 1988]. In general, eq. 2.19 can be replaced with:

$$w_{k+1,i} = w_{k,i} - \alpha \frac{\partial e_t}{\partial w, i} \tag{2.20}$$

where w represents the parameters of a controller. This may allow the learned controller to handle arbitrary curves if the controller accepts desired state as an input and its parameter changes are appropriate across all curves. Unfortunately, the choice of parameters in eq. 2.20 is usually a tradeoff between handling arbitrary curves and being able to concentrate on specific curves to run them open-loop. Since this learning rule is applied once for the error at each time step without considering the contribution of previous controls to the error, use of it places a system in the Closed Loop box of Fig. 1.1 and the upper right quadrant in Fig. 2.8.

It is also possible to obtain gradient information through a learned model of the plant. Jordan and Rumelhart use a back-propagation network to learn a forward model of the plant [Jordan and Rumelhart, 1992]. Then the derivatives relating controller parameters to performance error can be found by propagating the plant output back through the plant model with the usual neural network back-propagation equations. These are used to train a controller with a learning rule like eq. 2.20.

Jordan and Jacobs show how a forward model with only partial information can still be used to train a good controller [Jordan and Jacobs, 1990]. They look at the problem of stabilizing an inverted pendulum (a restricted form of the problem in Chapter 4). Rather than making a full mapping from current states and actions onto future states, they only learn a prediction from current state and actions to "time until failure" under the current control system. Time until failure is the number of steps until the pendulum goes past a certain angle. From that mapping they compute the derivative of time to failure with respect to control action and use a learning rule that seeks to increase the time to failure. This scheme learns to stabilize the pendulum by modeling the combined dynamics of the plant and controller together. This system considers the future effects of its actions, but only a learns a single trajectory (keeping the pendulum stationary and upright) so it belongs in the lower left quadrant of Fig. 2.8.

Gradient descent is dangerous if the surface has many local minima as they will prevent the system from converging to the best controller. All of the experiments described earlier in Section 2.4 were designed so there were no sub-optimal local minima. Sanger [Sanger, 1994] considers parameterizing tasks with a difficulty parameter as a method of removing undesirable local minima from learning control problems. One example is curve tracing problems where the desired speed can be adjusted. At slow speeds the dynamics are simple and a linear controller works well enough. At higher speeds the dynamics become more complex. He adopts a learning rule in the form of eq. 2.20 and learns as the desired curve tracking speed is gradually increased. The result is that his system learns to track a curve at high speed even though the learning rule would fall into a sub-optimal local minima if applied immediately to the problem at that speed. He proves convergence of his algorithm under assumptions about the smoothness of the plant, the stability of the plant, and the boundedness of errors generated with changes in the difficulty parameter. Sanger's algorithm belongs in the lower left quadrant of Fig. 2.8 because he must train the system for each value of the difficulty parameter and he does not accumulate the ability to control the system over the whole range of values. When his training is complete for high speed tracking, the controller will no longer produce accurate low speed tracking. Chapter 4 describes an algorithm for attacking problems that can not be represented as optimization problems with single minima. It uses the idea of learning to track curves at high speed by starting with low speed, but does so on an unstable plant that does not fit the assumptions of Sanger's work.

2.4.4 The cost of sensing

Most of the work in control theory assumes that sensors are present and functioning at all times. In Chapter 5 we look at how to learn skills and gradually give up dependence on sensor information so the sensors may be used for other purposes in a larger system. This idea implies a cost of using sensors and several authors have looked at it. Hager discusses the cost of sensing from the viewpoint of sensor fusion [Hager and Mintz, 1991; Hager, 1992]. He attempts to recognize or model unknown objects and chooses sensing actions that maximize the amount of information obtained for the amount of cost incurred. Rimey and Brown use decision theory to choose the sensing actions that will answer a question about a visual scene with minimal cost [Rimey and Brown, 1994]. Wixson and Ballard use cost considerations to guide visual search for objects [Wixson and Ballard, 1994].

Tan [Tan, 1992; Tan, 1993] includes sensing with other actions in the framework of reinforcement learning. One task is to grasp an unknown object and his choice of actions includes various sensing and grasping actions. Each action has its own cost (fixed) and expected payoff (a function of the information obtained so far). At the simplest level, the choice is between obtaining additional information through sensing, or trying to pick up the object with the current best estimate of which grasp will work.

Langley looks at the cost of sensing in a more traditional control scenario [Langley *et al.*, 1994]. He has a control system with a cost assigned to sensing and another assigned to tracking error. As the controller operates, the optimal sensing interval can be computed as a function of these costs, and the characteristics of the plant. Since no learning is included the sensing interval is assumed to be fixed. In Chapter 5 we will see how it can be increased as learning improves control.

2.4.5 Transfer of skill knowledge

Often humans can contribute knowledge about how a skill should be performed, but it is not in a form appropriate for the robot controller to use. For example, one of the benefits of fuzzy controllers (see [White and Sofge, 1992] for an introduction) is that it may be simpler to construct control laws using common sense rules than with other methods. Handelman proposes a system that converts rule-based fuzzy controllers into appropriate gains for PID controllers [Handelman *et al.*, 1991].

Gelfand consider several tasks that are easy to perform with the use of visual feedback, but assume it is preferable to do the tasks using only proprioceptive feedback [Gelfand *et al.*, 1992]. His system is run under visual feedback while a second controller learns the correct mapping using proprioception. Eventually, the visual feedback controller can be turned off. We make a similar assumption in Chapter 5 where we attempt to reduce dependence on sensor information.

2.4.6 Other related work

"Active learning" is a concept covering any learning problem where the acquisition of training data is expensive, but controlled by the learning system. An example of this work on classification problems is [Cohn *et al.*, 1994b] and an example on continuous function approximation problems is [Cohn *et al.*, 1994a]. This work is an important mathematical contribution to learning, but can not be directly applied to robot skill learning because of a subtle difference in problem formulations. Assume there is a function mapping $x \to y$. Much of the active learning literature assumes a goal of estimating the y values for a set or range x values and the ability to query x values. Robot skill learning also assumes the ability to query x values (actions), but has a goal of finding the x values to produce a set or range of y values (results). In skill learning the goal space is not the same as the query space. This is particularly devastating if the mapping is redundant (the query space has much higher dimensionality than the goal space).

The way a problem is represented can have a large effect on skill learning [Schneider and Brown, 1993]. By altering the representation of the set of actions it is possible to expand (or shrink) the volume of "useful" actions and thus make learning easier (or more difficult). Even choice of control signal primitives is important. In a throwing task kinematic control via joint velocities produces significantly better results than dynamic control via joint torques.

A number of authors have looked at the problem of learning a "juggling" skill [Aboaf *et al.*, 1989; Rizzi and Koditschek, 1992; Rizzi and Koditschek, 1993; Sakaguchi *et al.*, 1993; Schaal and Atkeson, 1993b]. The goal is to bat a ball upwards with a series of hits designed to keep the ball from hitting the floor for as long as possible. This work is closely related to the ball throwing experiments presented in Chapter 3 since each hit is an open-loop action parameterized by the current state of the ball and the desired place to send it (an arc that will bring it back near the center of the paddle for the next hit). Most of this work uses some modeling information and a gradient descent learning rule to adjust to its inaccuracies.

Of particular interest is the work by Schaal and Atkeson. They devise systems that juggle completely open-loop by constructing paddles that automatically redirect the ball back toward a stable periodic trajectory with each hit they make. Similarly, McGeer [McGeer, 1990] designed a walking robot that is stable without any sensing or actuation. It walks stably downhill powered only by gravity. Also, it is possible to stabilize an inverted pendulum open loop by driving it sinusoidally up and down [Acheson and Mullin, 1993]. These open-loop systems are interesting because they are robust without sensing. In each case the authors spent significant effort designing and analyzing their systems. These tasks are also possible applications for the open-loop learning discussed in the following chapter.

3 Open Loop Motor Skill Learning

Open-loop motor skills are those for which no sensory information is used during the execution of a task. The controller receives feedback only between trials for its learning algorithm. Before a trial begins, the controller is presented with a desired result and a current state. From that information it must compute all the desired control signals necessary to obtain the result. Then the trial is run with the controller sending control signals to the plant. The results of the execution become available at the end of the trial and are used in the learning process. The block diagram of an open-loop skill learner is shown in Fig. 3.1.

There are several advantages to performing skills open-loop. The most important is that execution does not depend on sensors. Sensors are a problem when they are inaccurate, noisy, or have too much delay or too little bandwidth. Each of these problems limits how fast the plant can be driven through its state space. Open-loop learning has the advantage that it considers the controller's performance on an entire task.

There are also several disadvantages to performing skills open-loop. The first is noise in the plant. If it is impossible to control the plant in a way that the same sequence of control signals always produces (nearly) the same output, sensors are required to get reasonable performance. The second is that the dimensionality of the action space is very large for open-loop skills. This is because a *sequence* of control signals makes up the space of possible actions. Its dimension increases linearly with the length of

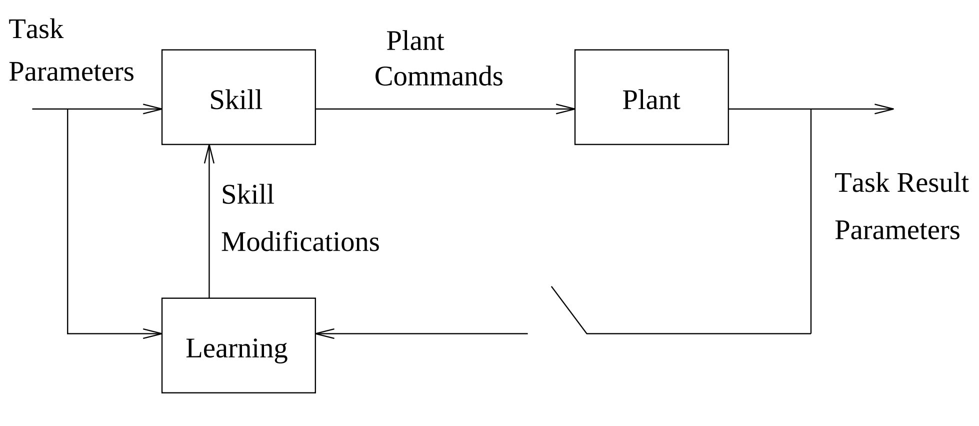

Figure 3.1: Block diagram of open-loop motor skill learner

the sequence (dimension is the product of the number of time steps and the number of controls per time step). The lack of ability to search large action spaces is often what prevents skills from being done open loop. In this chapter we present an algorithm for efficient search of high-dimension action spaces.

3.1 The open loop skill learning problem

Fig. 3.2 shows the open-loop skill learning problem. It may help to visualize a specific open-loop skill such as throwing a ball. The learning problem for a robot thrower is to find the control sequences for the arm that can hit targets placed in front of it at any of a set of ranges. The high-dimension action space is represented here as a 3-d space. It is constrained in that each control signal may not exceed a maximum and a minimum value.

The task result space is represented as a line because the first throwing example uses a one-dimensional result (the distance the ball travels). The result space may have any dimension, but we assume it to be lower dimension than the action space. Note that throwing a ball at a single target (as in [Aboaf *et al.*, 1988; Branicky, 1991]) represents a zero dimension result space and corresponds to our concept of a task, but not a skill. We assume that a memory-based function approximator (in fact, table lookup with interpolation) is used to store the learned skill. Each thin arrow in Fig. 3.2 represents a single entry in the table that associates an action with a result. The robot can only derive the *Forward Plant Mapping* by executing an action and observing its result. Our goal is to compute the *Robot Skill* so we may specify a desired result and find the action to produce it.

We rate the controller's performance on its range of operation, accuracy, and control effort. All three of these measures are common in the field of optimal control, but many others are possible. We choose the first two because they are obvious goals for a throwing robot, and the third because it helps us obtain the first (demonstrated later). The range of operation is measured as the size of the task result space the controller can cover. Note that limits on the control signal magnitudes are usually the main factor in limiting range of operation. The control effort is measured as the average sum of squared force needed over the range of operation. If we assume a one-dimensional task result space and one scalar control signal per time step, the cost functional takes the form:

$$J = \alpha_0(y_{max} - y_{min}) + \int_{y_{min}}^{y_{max}} \alpha_1 \sum_{i=0}^{T} (u_i^y)^2 + \alpha_2 (y - y^*)^2 dy \tag{3.1}$$

where y_{min} and y_{max} mark the range of operation, T is the number of actions in the sequence, u_i^y is the control signal at time i used to produce output y, y^* is the actual resulting output, and the αs are relative weights. We set α_0 negative and α_1 and α_2 positive so the object is to minimize J.

The mapping from an m-space of actions to an n-space of task results is redundant if $(m > n)$. The unique inversion of a redundant mapping requires a cost function such

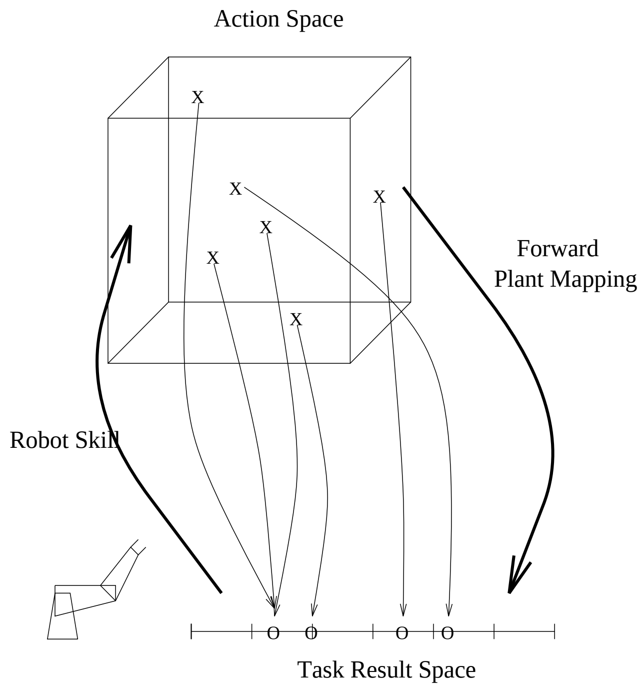

Figure 3.2: Diagram of open-loop motor skill learning problem

as eq. 3.1 that must be optimized. Note that computing the robot skill is a search problem where the goal is to find the optimal inverse for a redundant mapping. We are searching for the curve (more generally, subspace) in the action space that maps onto the task result space with the minimum value of J and that optimal subspace may be discontinuous. Therefore, it is qualitatively different from search problems where a single (minimally valued) point is sought.

It is necessary to search the entire action space to guarantee finding the optimal result whether it is a subspace or just a point. For a brute force search at a given level of quantization along each dimension, the required search time is exponential in the number of action space dimensions. We consider problems where it is impossible to guarantee an optimum (most high-dimension learning problems with insufficient prior modeling information are), but we would like to improve performance as much as possible through learning. Each trial on the real robot is expensive so it is important to choose actions intelligently. The following section describes an algorithm for open-loop robot skill learning.

3.2 The guided table fill in algorithm

The Guided Fill In (GFI) algorithm (reported in [Schneider, 1994]) is a modification of the SAB control algorithm given in Moore's thesis [Moore, 1990]. He was also concerned with the efficient search of action spaces, but did not specifically address the issues of inverting a redundant action to task result mapping. Because of the time constraints

Fields of a Table Entry			
p^{act}	p^{res}	p^{eff}	p^{good}
action sequence	task results	control effort	relative goodness value

Table 3.1: Table entry for open-loop skill learning. The table is indexed according to p^{act} for candidate evaluation and according to p^{res} for candidate generation and normal operation.

in real systems, it is often impractical to search the entire plant command space. Skills involving redundant mappings have an increased need for search efficiency because it is especially undesirable to search a high-dimension action space in order to compute the skill for a low-dimension task result space.

The result of each system execution is stored in a table. The fields of each entry are listed in Table 3.1. P^{act} is the action sequence tried and p^{res} is the result in task space. P^{eff} is a scalar measure of the control effort required by the action sequence. P^{good} is a goodness value for the entry (its computation is described below).

Guided Table Fill In is summarized in Fig. 3.3. The first step is one of several opportunities to incorporate a priori information about the task or the robot. Existing models or teachers can determine several points from the plant command space to become the first entries into the table. Random choice is a worst case, but possible.

There are two parts to a table entry's evaluation. First (done only once): given a point in plant command space, the system executes the corresponding action (p^{act}) and observes the output (p^{res}) and its control effort value (p^{eff}). The task output parameters determine where the point is recorded in the table. Second (executed once each iteration): compare each point against its neighbors in the table (nearest points in the output space). A point's goodness (p^{good}) is the percentage of neighbors whose effort value is worse than its own.

Step 3 randomly chooses a goal from part of the task result space. There are better ways to select a goal that will be discussed later. Step 4 generates several candidate actions to accomplish the goal task result. Some of the actions are generated by local random modifications to existing "good" points. The table is searched for the entry that best accomplishes the desired goal considering both accuracy and control effort. The action for that entry is altered with small random changes to some of its values. Several alterations are done to produce a set of candidate actions.

Moore also advocates generating some actions from a uniform distribution over the entire space of actions. The purpose of these candidates is to keep the learner from getting stuck in local minima. Experiments with redundant mappings showed that these global random candidates were not useful. Moore's algorithm always performed best when its parameters were tuned such that none of these candidates were ever executed. The reason is that totally random points in a very high-dimensional action space virtually never produce a useful result. However, it is still necessary to generate candidate actions far from the existing set of actions in the table. To do this we can

Guided Table Fill In Algorithm

1. Initialize table using an approximate task model, a teacher, or random choice.
2. Evaluate each point in memory according to its control effort. Assign a "goodness" value based on a comparison of a point's control effort value to its neighbors' (in task output space) values.
3. Randomly choose a goal from the desired portion of the task parameter space (optionally, use current range of operation to bias the distribution of chosen goals).
4. Generate candidate actions from the robot action space using both of the following:
   - Find the table entry whose penalty function is lowest for the desired goal (considering accuracy of task result and control effort). Make random modifications to the action sequence of that entry.
   - For an n-d task space choose $n+1$ "good" samples from the table and interpolate or extrapolate to produce an action for the new goal. "Good" is defined in terms of the evaluation at step 2.
5. Evaluate the probability of success for each of the candidate points.
6. Execute the action with the highest probability of success and store the results in the table (optionally, readjust the desired range of operation).
7. Goto 2.

Figure 3.3: Guided Table Fill In Algorithm. See text for a detailed explanation

take advantage of the fact that we are searching for a curve in the action space [1]. We use points on the current "best" curve to suggest new points in unexplored regions of the space. This is done by using linear combinations of existing "good" points in the table. Several sets of points are chosen randomly with the "best" points having a higher probability of being chosen. Interpolation or extrapolation is done from the chosen points to generate new actions. These actions may be in unexplored regions of the action space, but are likely to be more useful than completely random actions because of the way they were generated.

Step 5 evaluates the probability of success as Moore suggests [Moore, 1990]. For each candidate action, p^{act}, find the table entry whose action, p^{act}_{near}, is nearest the candidate action. The probability of success for each dimension of the task result space is determined as follows:

$$Pr_{success}(p^{act}) = erf\left(\frac{goal - p^{res}_{near} + \tau}{\mid p^{act} - p^{act}_{near} \mid}\right) - erf\left(\frac{goal - p^{res}_{near} - \tau}{\mid p^{act} - p^{act}_{near} \mid}\right) \tag{3.2}$$

$$erf(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2} dt \tag{3.3}$$

The probability computation is shown pictorially in Fig. 3.4. The standard deviation of the distribution is determined by the distance between the candidate action and its nearest neighbor in the action space. The distance between the goal and the task result of the nearest neighbor determines what part of the normal curve is integrated. The motivation for this probability evaluation is that actions will be ranked from best to worst according to:

1. The candidate is confidently predicted to succeed.
2. The candidate is unconfidently predicted to succeed.
3. The candidate is unconfidently predicted to fail.
4. The candidate is confidently predicted to fail.

When several task result dimensions are considered, a probability is computed for each dimension and the product of them is the probability for the whole task result. When considering redundant mappings, reducing control effort is important so we make it a goal too. A goal effort is selected (usually a constant, unattainable, low value) and control effort is considered to be another dimension in the above computation.

Finally, the candidate with the highest probability of success is executed at step 6. The results of the execution are recorded and a new table entry is made. Steps 2-6 are iterated during the training process.

At this point it is important to highlight the differences between GFI and Moore's original algorithm. There are two key ideas that make the GFI algorithm work well for

[1] This assumes that the curve is continuous. Linear combinations are useful in areas where the curve is continuous, but seem less likely to bridge gaps between unconnected segments.

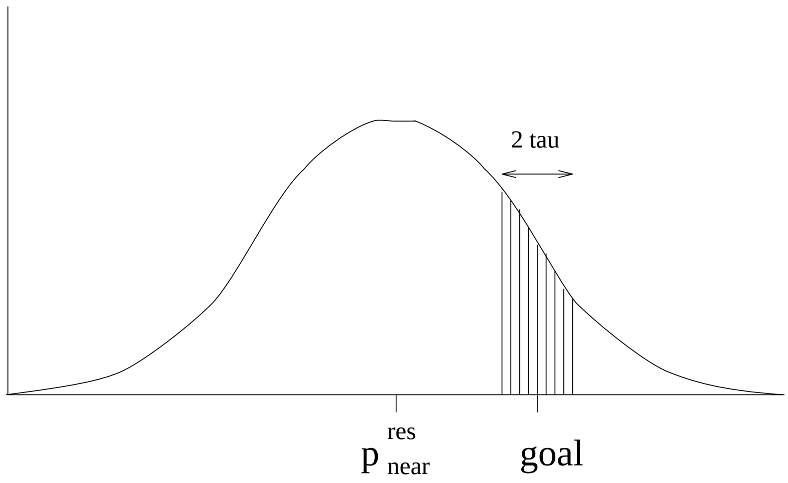

Figure 3.4: Candidate probability of success

inverting highly redundant mappings. First is replacing the global random candidates with linear combinations of existing "good" points. This technique explores new regions of the action space in a way that is much more likely to produce useful trials under the assumption that the optimal curve is continuous and has reasonably low curvature. Second is the inclusion of control effort as a dimension of the task result space. This causes the algorithm to prefer points that use low control effort. Low control effort by itself is often important in skill learning applications. More importantly, however, there are usually limits on the control signals that can be sent to the plant. Low control effort actions tend to stay away from those limits and thus are more useful in future extrapolations.

Action selection during performance mode

Until now we have only considered how the system should choose its actions while learning. To be complete we must provide an algorithm for choosing actions when the system is being asked to perform at its best. Redundant mappings raise extra concerns for action selection. It is standard practice to interpolate between nearest neighbors in table lookup learners. However, not all data stored in the table represents a "low effort value" function mapping, so additional factors must be considered when choosing points from the table. In performance mode the task goals are determined by an external source. The Guided Fill In learning algorithm does not worry about the accuracy of output obtained from each trial – "it is only practicing" and every execution helps fill the table. By definition though, accuracy is important during normal operation. Unsurprisingly for highly nonlinear systems, our experiments have shown that extrapolation often leads to wildly varying task results (good for filling in the table, but bad for performance). Therefore it is best to look for a set of points from which the task goal can be interpolated. Failing that, it is best not to extrapolate at all and simply choose the closest single point in the task parameter space instead.

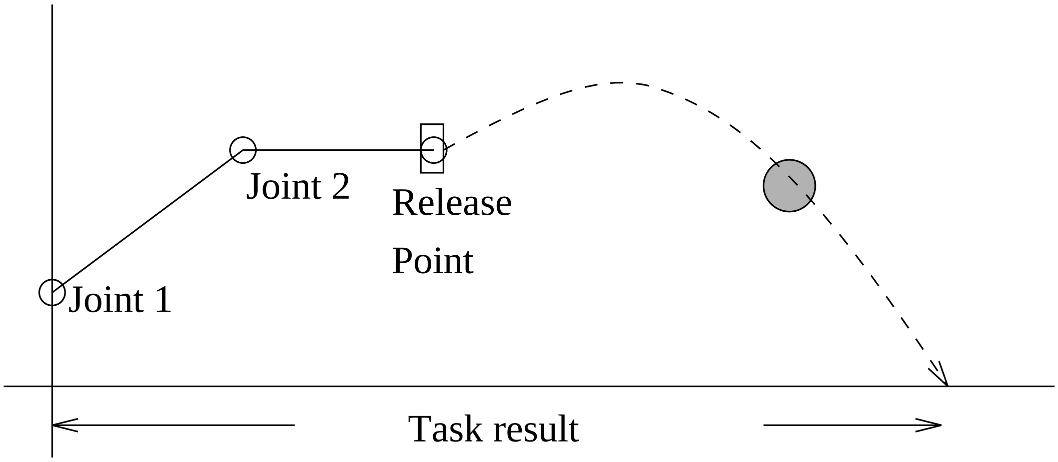

Figure 3.5: One-dimensional throwing task

Choosing the best set of points to interpolate between is also a non-trivial operation. Consider a one-dimensional output example. Choosing the nearest two table entries whose result is on either side of the desired result may not be a good idea. Suppose one of the points has low control effort and is in one portion of the action space while the other has high control effort and is in another. Interpolating between them will probably yield a point with medium control effort, but the task result is unlikely to be near the desired one. Interpolating over a great distance in a complex space is risky. Therefore, it seems reasonable to consider all pairs (or sets in higher-dimensional output spaces) and rate them on how near their output values are to the desired output *and* how near they are to each other in the action space. In the experiments that follow we use this action selection algorithm when evaluating the progress made by the skill learner.

3.3 Simulation experiments with ball throwing

The Guided Fill In algorithm was tested on a robot throwing skill. The task goal is a vector describing the position of a target and the actions are sequences of joint velocities for a throwing robot. The robot and the forces affecting the ball's flight after it leaves the robot make up the plant. We refer to skills as n-dimensional where n is the number of parameters in the output space of the task: thus a 2-d throwing task has a target lying in a plane, such as the floor. A 1-d throwing task considers only the distance the ball travels from the base of the robot.

Experiments were simulated for 1-d and 2-d throwing tasks [2]. In the 1-d task the robot consists of two joints in a vertical plane (Fig. 3.5). The control signal is a sequence (also called a trajectory) of joint velocities to be executed over 6 time steps, thus making a total of 12 input command parameters. At each time step the velocity at each joint

[2]Although the algorithm is tested in simulation, it is designed for use on real systems and we keep the assumption that data acquisition is expensive by limiting the number of learning trials.

is set to a new value. The single output dimension is the distance the ball lands from the base of the robot. In the terms of table lookup, a 12-d space must be searched to fill in a 1-d table. The 2-d throwing task is done with a three joint robot configured like the first 3 joints of a PUMA 760; a rotating base is added to the 1-d thrower. The additional joint yields an 18-d search space. The two task parameter dimensions are the x and y (or ρ and θ) coordinates of the ball's landing position on the floor.

The cost function includes the approximate amount of energy required to execute the throwing trajectories and the average task output error. For these experiments we will consider a fixed range of operation and not use it in the cost function. In the following equation for the two joint arm, the plant command vector (trajectory) is considered to be an array, $\mathbf{x}$, with one dimension for the robot joints and the other for time steps.

$$J = \alpha d^2 + \sum_{j=0}^{1}(x_{j,0}^2 + \sum_{i=1}^{5}(x_{j,i} - x_{j,i-1})^2) \tag{3.4}$$

where i ranges over the time steps, j ranges over the joints, d is the Euclidean distance between the target and where the ball actually hit, and α is a weight to control the relative importance of accuracy and trajectory smoothness. The term "approximate amount of energy" is used because the function actually sums the squares of accelerations rather than the torques necessary to produce the trajectory. The approximate energy measure has a second purpose. The simulated robot has limits on its joint velocities and the metric tends to prefer trajectories that stay away from those limits. Later, the average value of the cost function over the task parameter space will be referred to as the *performance* and the two terms will be referred to separately as *error* and *effort*.

For some skills, the dimension of the plant command space can be reduced [Schneider and Brown, 1992]. In this example, the positions and velocities of the joints as the ball is released completely determine where it will land. Furthermore, there is a closed form solution (presented in Appendix A) for the lowest energy control sequence that produces a given set of final joint positions and velocities. Therefore, the plant command space for the throwing task can be reduced to a vector representation of the joint release positions and velocities. In the 1-d throwing described above, the input space is reduced to four dimensions. Similarly, the 2-d throwing input space is reduced to six dimensions. This reduction in the search space is used in all the simulation experiments that follow. Note that this reduction is not necessarily possible for other common cost functions (e.g. sums of torques) or other types of robots (e.g. flexible joints or links).

3.3.1 Random fill in experiments

A random fill in algorithm (called RFI) was implemented to provide a baseline from which to compare algorithms. At each step a new random action is formed by choosing ending joint velocities and positions from a random, uniform distribution over the range of valid values. The new command sequence is executed and recorded in the table. Retrieval from the table during performance mode is done as described earlier by finding two (three) points for linear interpolation in the 1-d (2-d) case. Extrapolation is never

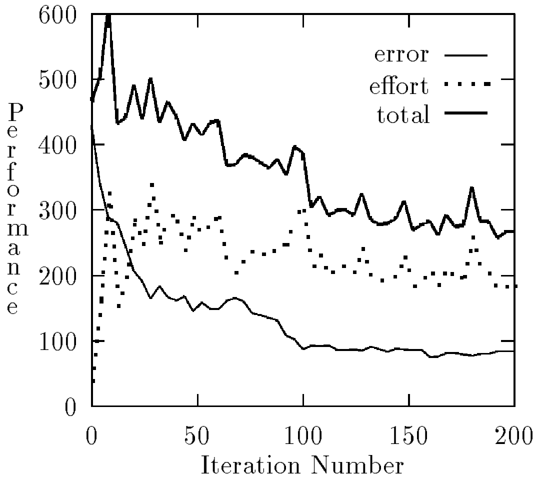

Figure 3.6: 1-d RFI: range 2500

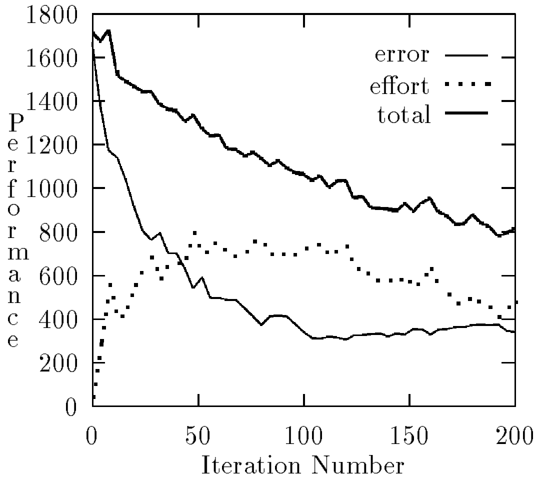

Figure 3.7: 1-d RFI: range 5000

used for data retrieval. If no points can be found for interpolation, the nearest point in the task parameter space is chosen. When interpolation is possible, a scalar value is given to each possible pair (triple) of points to determine which should be used. It includes terms for the distance of the points from the goal in task result space, the average effort associated with each point's command sequence, and the distance of the points from each other in the action space.

Figs. 3.6 and 3.7 show the results of experiments with RFI for the 1-d throwing tasks. The desired range of operation is 1300-2500 mm and 1300-5000 mm respectively (the robot's arm has a fully extended length of 1000 mm). The graphs represent averages over 20 separate runs. The x axis is the number of learning trials, or the number of action-result pairs that can be placed in the table. The y axis is an evaluation of the robot's progress learning the skill according to eq. 3.4. To evaluate the robot's ability, ten targets evenly spaced in the desired range are attempted and the average effort and accuracy are recorded.

Fig. 3.12 shows the results of random fill in with a 2-d throwing task. The x-range is 1300-2500 mm (as with 1-d throwing) and the y-range is 0-1200 mm. Only throws on the left side (positive y values) of the line representing 1-d throws are considered since those to the right (negative y values) can be accomplished by negating the joint velocities of the first joint. One characteristic of the graph is that the total cost metric appears to get worse as the learning progresses. This happens because the robot is capable only of short throws initially (it is given sample short-throw trajectories to start the learning). It uses those trajectories even when long throws are requested and pays the penalty in accuracy (because it refuses to extrapolate). As it finds trajectories to throw greater distances, it uses them and accuracy is improved. However, these trajectories require greater effort thus causing the overall performance to look worse. This initial decreasing error and increasing effort is a characteristic of several graphs presented here.

3.3.2 Results with Guided Fill In

The 1-d throwing experiments of the previous section were repeated with Guided table Fill In (GFI). The evaluation and action retrieval methods are the same as for RFI. Five initial actions capable of throwing distances from 1370 to 1450 were given in Step 1 [3]. Figs. 3.8 and 3.9 show the results. The GFI executions attain values of 55 and 254 after 200 iterations compared to 268 and 818 in the RFI runs. A comparison of the graphs shows that GFI gets most of its improvement in the first 50 trials and thus performs its learning much quicker than RFI.

Further improvements can be made when using GFI for longer distances. "Automatic range expansion" is a modification of the algorithm that allows it to choose its range of attempts according to its current abilities. Step 6 optionally updates the current range of operation achieved by the system. Step 3 calls for choosing goal task parameters within some desired range. That range is set to be the current range of achieved results plus some distance farther. A parameter controlling how much farther trades off how quickly the range of operation will be increased with how many trials will be spent searching for low-effort actions within the current range. Fig. 3.10 shows standard GFI for a range of 10000 and Fig. 3.11 shows the results of experiments using automatic range expansion to a distance of 10000. The additional set of points represent the maximum distance attained (divided by 10 to fit in the graph). Shorter distance throws are easier (there are more trajectories in the command space that accomplish them). The algorithmic modification allows the robot to learn the easy part first and then move on to more difficult throws. As the graph shows, it reaches a distance of 10000 by the end of 200 iterations and attains a final performance value of 1072 (compared to 1448 with standard GFI). Traditional engineering practice calls for the range of operation to be pre-specified for system design. However, when dealing with highly complex plants this may be difficult. Therefore, this modification is also important when a robot system designer does not know the range of performance that can be achieved, but wishes to maximize it.

The 2-d throwing experiment was also done for GFI. The x distance was 2500 mm and the y distance was 1200 mm. Its results are in Fig. 3.13. GFI is significantly better than RFI for the 2-d task. It gets a final performance value of 155 compared to 1247 for RFI (Fig. 3.12). This result shows that performance gains can also be seen in tasks of higher dimensionality.

A more revealing statistic is the number of runs RFI requires to reach the performance attained by GFI. When skill learning is attempted on real robots, the number of executions required becomes extremely important. For a 1-d range of 2500 RFI requires an average of over 100000 iterations to reach the level GFI attains after only 200. For a 1-d range of 10000, RFI requires 10000 iterations to match 200 of GFI with automatic range expansion. The larger discrepancy between RFI and GFI on the shorter task shows the benefits of using a fill in algorithm that is guided by its desired range of operation vs one that randomly tries any valid trajectory. GFI is able to concentrate its trials on the portion of the trajectory space that throws in the range 1300-2500.

[3] For a fair comparison these five were also given prior to learning in the RFI experiments.

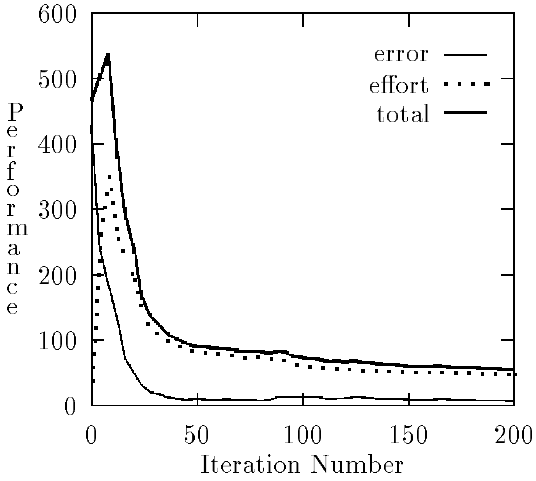

Figure 3.8: 1-d GFI: range 2500

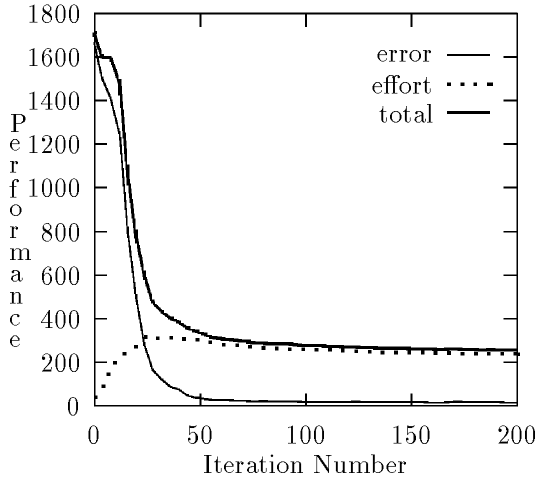

Figure 3.9: 1-d GFI: range 5000

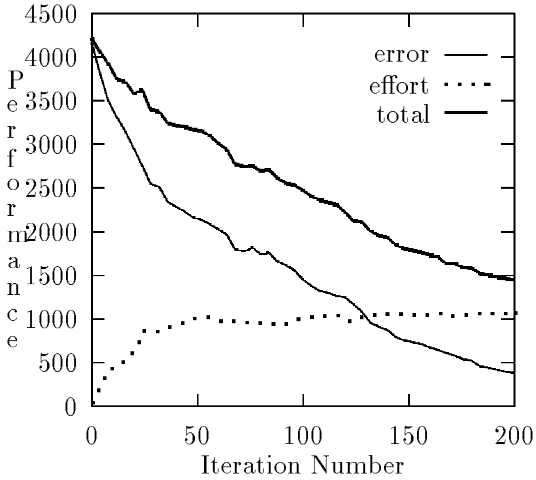

Figure 3.10: 1-d GFI: range 10000

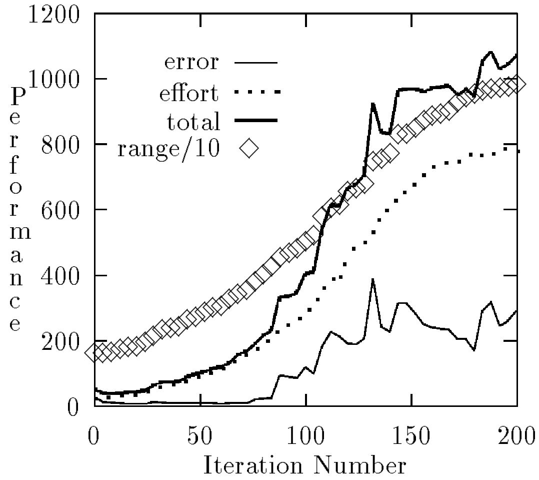

Figure 3.11: 1-d GFI: auto range expand

Task	Optimal value	GFI value 200 iterations	RFI value 200 iterations	RFI iterations to equal 200 GFI
1-d 2500	24.4	55.0	267.6	more than 100000
1-d 5000	164.6	254.4	817.7	10000
1-d 10000	467.1	1072.3	2443.8	10000
2-d 2500,1200	39.2	155.5	1246.9	more than 100000

Table 3.2: Summary of experimental results: Optimal is numerically estimated. GFI is the performance of the new algorithm after 200 iterations using automatic range expansion when it produces improved results (automatic range expansion made no significant improvements for ranges less than 10000). RFI is from traditional table lookup with random trials. The last column indicates how many trials RFI needs to equal GFI's performance.

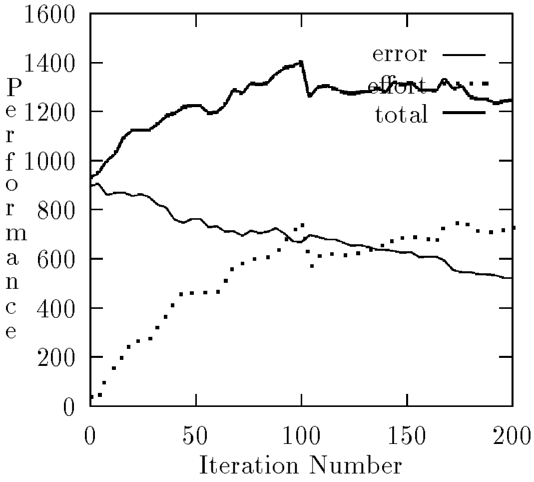

Figure 3.12: 2-d RFI: range 2500,1200

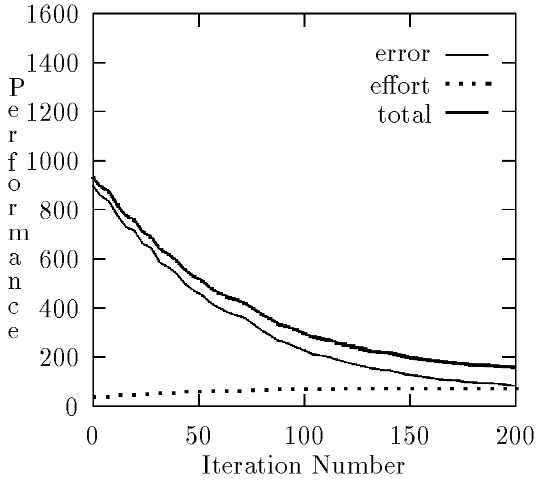

Figure 3.13: 2-d GFI: range 2500,1200

Similarly, RFI for the 2-d case required more than 100000 iterations to equal the GFI performance with 200 iterations.

The actions attempted with 1-d GFI and RFI throwing examples show how the algorithm works. Since the search space contains the ending positions and velocities of two joints, the skill learner is looking for a curve through a 4-d space. Figs. 3.14 through 3.17 are projections of the 4-d space onto the plane defined by the ending joint velocities (ending joint position information is lost). Fig. 3.14 shows that the distribution of attempted points is fairly uniform throughout the search space when using RFI. The lower density at the left is caused by a higher number of invalid trajectories there. Fig. 3.15 shows a projection of the optimal curve for a range of 300-5000. A description of how the optimal curve was numerically estimated is in Appendix A.

Figs. 3.16 and 3.17 show the points attempted by GFI at two different ranges. The figures show that GFI concentrates its trials in a small subset of the search space. The optimal curve verifies that the GFI trials are concentrated in a good portion of the space. The area near the center where the two GFI graphs have points in common is where the five initial points are located. A comparison of the two GFI graphs shows the algorithm starting from the initial points and working toward the optimal for the desired range.

3.4 Experiments on a PUMA

The GFI algorithm was tested on a real robot learning a 1-d throwing skill. We use a PUMA 760 with a flexible link (a metal meter stick) attached to it. At the end of the meter stick a plastic cup is attached to hold a tennis ball for throwing experiments. The release point of the ball is not directly controlled. The ball comes out whenever it is moving forward faster than the cup. A CCD camera watches the landing area and determines where the ball lands with respect to the base of the robot. Most of the

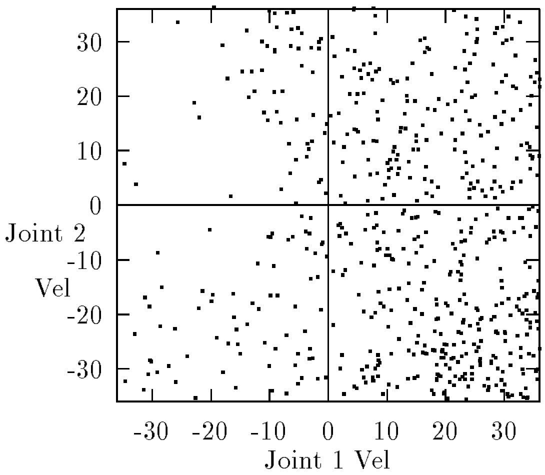

Figure 3.14: Projection of input space points attempted: RFI range 1300-5000

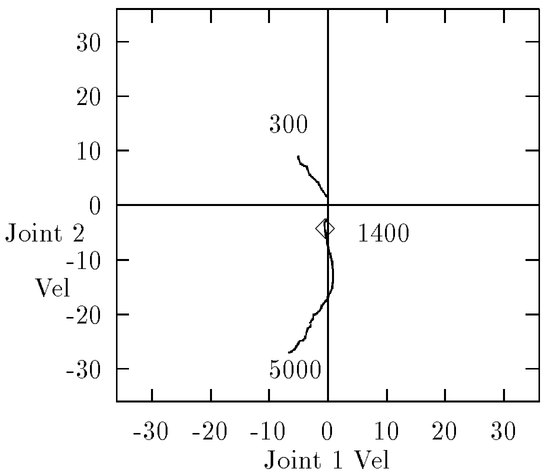

Figure 3.15: Projection of input space points: Optimal 300-5000

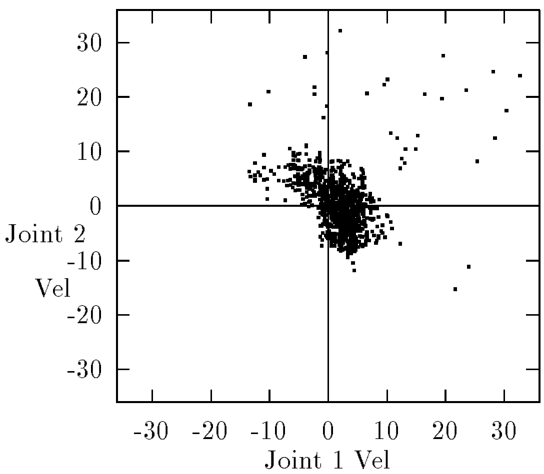

Figure 3.16: Projection of input space points attempted: GFI range 0-1500

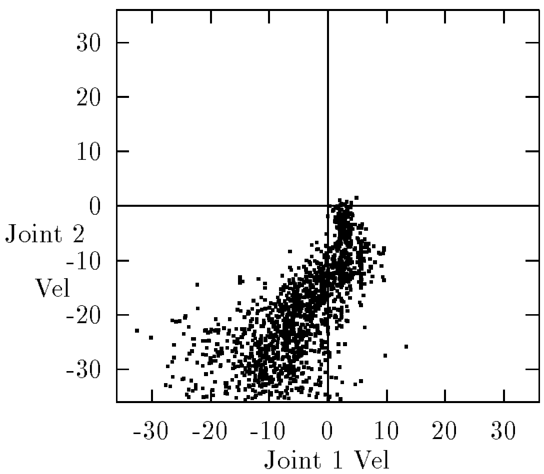

Figure 3.17: Projection of input space points attempted: GFI range 1300-5000

parameters of the experiment were set the same as the 1-d throwing done in simulation. Two joints of the robot are controlled by specifying joint velocities over six 200 ms time steps. The low-level controller, RCCL, interpolates between them to produce a smooth velocity profile. The control effort function is the same (eq. 3.4). As in the simulation, the effort function prefers trajectories that are far from the robot's physical limits. The reduction from 12 dimensions to 4 is not possible because of the flexible link. Therefore, these experiments test GFI in a much larger action space than the simulation did.

The GFI algorithm was given four sample actions that resulted in throws ranging from 143 cm to 164 cm. The velocity profiles of the samples are shown in Fig. 3.19. The samples were generated by making all four combinations of the two trajectories on the two joints (joints 3 and 5) used for throwing (i.e. joint 3 and 5 execute trajectory A from Fig. 3.19, joint 3 executes A while joint 5 executes B, etc.). Automatic range expansion was used because that option performed the best for 1-d throwing in simulation and because it was not possible to determine the robot's range of operation beforehand.

After 100 iterations the robot had learned to throw accurately out to a range of 281 cm. Its accuracy was good enough that it consistently hit a 2 cm screw placed upright anywhere within its learned range of operation. The same cameras that watch the landing position of the ball during learning locate the target during performance mode.

The most interesting result of the learning is the type of action found to work best for long throws. The four sample actions simply accelerate both joints forward. In contrast, the learned actions for long distances move the two joints forward, then stop or reverse direction, and then move forward with a large velocity. A sample velocity profile of this type is shown in Fig. 3.20. The motion stores energy in the flexible link by forcing oscillation in it. This results in a greater release velocity than can be achieved by simply accelerating the joints forward to a high velocity because each of the PUMA's joints are limited to a speed of 90^o/sec. Fig. 3.18 shows the throw produced by the velocity profiles of Fig. 3.20. The meter stick was manually darkened in each image to highlight its motion.

The significant aspect of the long throws is that they are *qualitatively* different from any that are given to the system at the start. The sequence of events that leads to the robot trying the action in Fig. 3.20 illustrates the GFI algorithm at its best. At iteration 15 the system was shooting for a goal of less than 200 cm (within its current range of ability). It chose an action created as a local random modification in step 3. That action had a significantly lower velocity at time step 2 for joint 3. The result was a throw for a distance of 164 cm with considerably lower control effort than any previous action for that distance. Later, at iteration 30, a similar thing happened with time step 3 of joint 5. The result was a low effort throw of 176 cm. Following that, the algorithm chose several actions that were generated by linear combinations of these unique actions. Large extrapolations through the new points created velocity profiles with the "whipping" motion observable in the sequence of 8 images. The cost function, small random modifications, and extrapolation all work together to find new, useful actions in unexplored portions of a high-dimensional space without having to resort to brute force search.

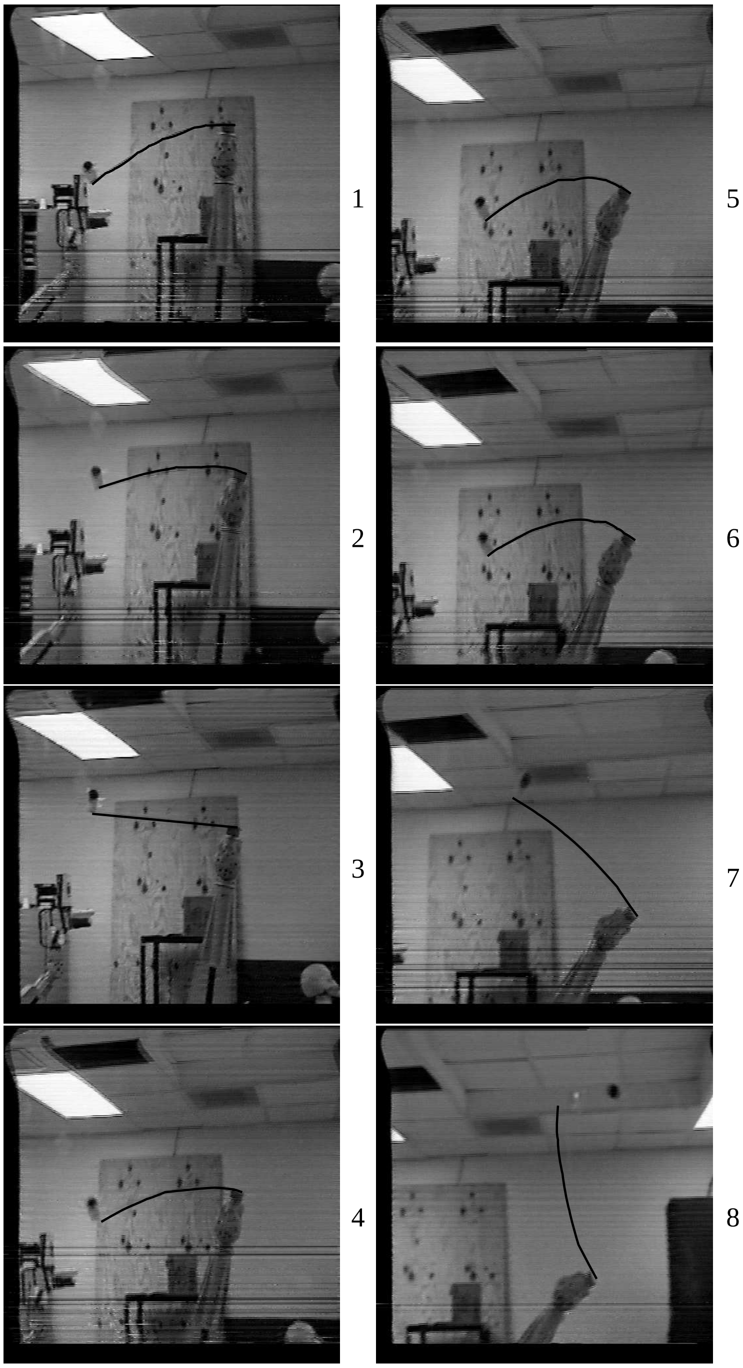

Figure 3.18: Whipping motion: the robot moves forward in 1 and 2, hesitates and reverses direction in 3, then resumes with maximum acceleration at 6, maximum achieved cup velocity at 7, and release at 8

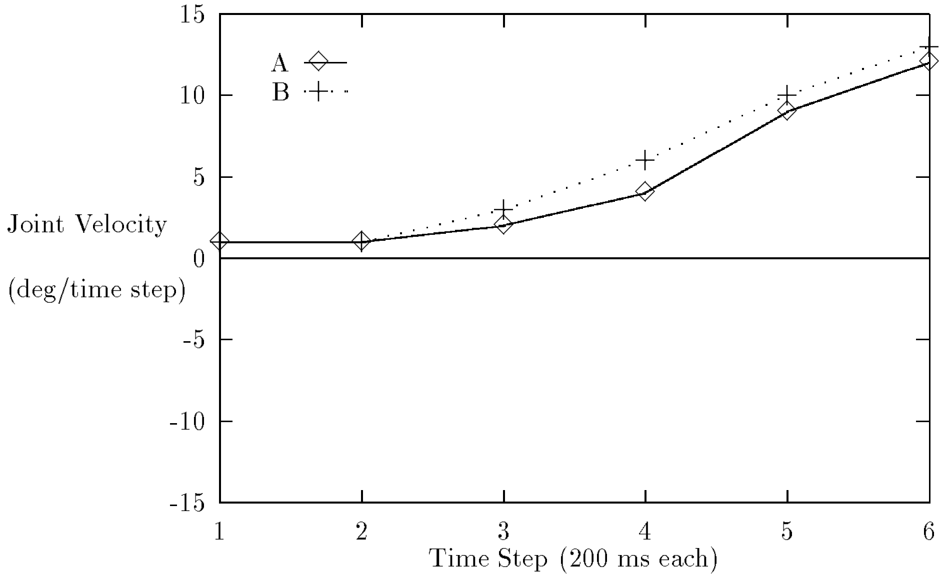

Figure 3.19: Initial velocity profiles

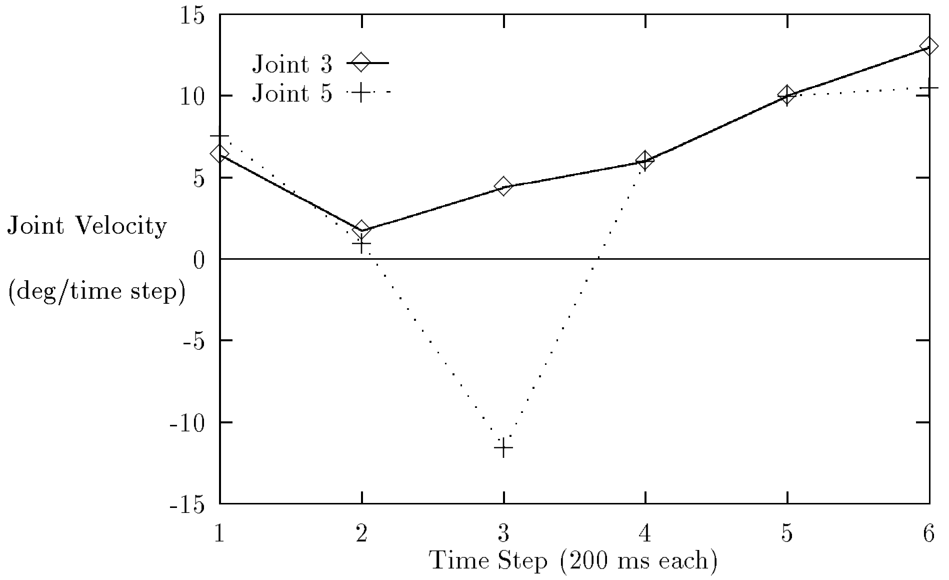

Figure 3.20: Long-throw velocity profile

3.4.1 Basis functions for the PUMA

Other work [Schneider and Brown, 1993] has shown the benefits of having a good basis function for a skill learning robot. Specifically, basis functions can restrict the action space search to only those areas that are most relevant to the task. The problem of finding, or automatically generating appropriate basis functions was noted as both interesting and difficult.

After observing the type of trajectories that were learned for long distances, we attempted to find basis functions that would speed up the learning. The long throws were produced by forcing oscillations in the flexible link. Therefore, it seemed reasonable that a sinusoid would make a good basis function. The first attempt called for controlling the robot by specifying a frequency and the two parameters of a ramp used to control amplitude. These three parameters (for each joint) replaced the six that had been used previously, but produced a joint velocity profile of the same time length. Unfortunately, learning with this basis function was poor. The robot never exceeded a distance of 200 cm. Several other modifications were attempted including using period rather than frequency, fixing the phase of the sinusoid to end at its full, positive value, and altering the way the amplitude was modulated. None of these produced better results.

A possible reason for the failure of sinusoidal basis functions is that it is more difficult to excite the system at its natural frequency using a single frequency velocity profile than one spanning a range of frequencies (such as step velocities). This failure demonstrates how difficult it can be to choose appropriate basis functions even when the desired system behavior is already known. Choosing them without knowing the desired behavior is at least as hard. When appropriate "dimension reducing" basis functions can not be found it is even more important to find a way to search the high-dimensional action space efficiently.

3.5 The use of models in open loop skill learning

Until now we have been assuming almost no model of the robot or task we are trying to learn. In this section we investigate what improvements might be made if a model is available. We have constructed a dynamic model to simulate the ball-throwing robot, with a view to replacing most of the actual robot cycles with simulated robot cycles [Schneider and Gans, 1994]. This makes the learning process faster and cheaper. There are two aspects of the simulation that make it nontrivial. The first is the large displacement deformation exhibited by the meter stick. The second is the lack of a formal release mechanism for the ball. We will discuss these two questions in turn. Following the description of our ball throwing model, we propose an algorithm that uses real executions to calibrate the model and simulated executions to replace some of the real ones.

3.5.1 Implementation

The following description is from [Schneider and Gans, 1994]. The meter stick clearly deflects beyond the range of linear beam theory, as can be seen in Fig. 3.18 taken from an actual experiment. We suppose the deformation to be that of a dynamic elastica (large deformation, small strain) and have adopted the segmented model of Gans [Gans, 1992], according to which the meter stick can be modeled by a set of rigid elements connected by linear torsional springs with spring constant NEI/L. N denotes the number of segments, E Young's modulus, I the section moment (so that EI represents the bending rigidity) and L is the effective cantilevered length of the meter stick (taken to be 90 cm). The model takes the form of 2N first order nonlinear ordinary differential equations in time:

$$A_{ij}\dot{\Phi}_j = b_i + f_i$$

$$\dot{\Theta}_i = \Phi_i$$

where the coupling matrix A and the vector b are functions of the segment angles Θ_i and their derivatives Φ_i, and the vector f denotes whatever inhomogeneous forcing is driving the system. The Lagrangian of this discrete model converges formally to the Lagrangian of the continuous elastica in the limit that $N \to \infty$, and tests of small oscillation frequency and buckling behavior show that the model is quite good for a relatively small number of segments.

We adopt $N = 3$ for our simulation studies, and we make some additional modifications. We suppose the meter stick to be rigidly attached to the end of the robot arm and the arm to have sufficient torque to drive whatever motions we desire. This compound hypothesis means that we can set $f_i = 0$ for all i and impose the motion of the first segment directly from the robot motion. This reduces the six-dimensional system needed for a three segment beam to a four-dimensional system by elimination of one Θ equation and one Φ equation. We further introduce a small dissipation to account for imperfect elasticity. This is done through a Rayleigh dissipation function in the Lagrangian, and the dissipation parameter is chosen to be proportional to N by analogy to the rigorous requirement for the spring constants. The magnitude of the dissipation is the only real adjustable parameter in our current model. It is determinable in principle from measurements of the decay of free oscillations of our meter stick. Later, in our learning experiments, we will also modify other parameters. This will make the result of simulated throws more closely match real throws at the expense of losing the precise physical interpretation of the parameters.

The ball is considered separately. We reduce it to a system of the form:

$$m\ddot{x} = f$$

$$m\ddot{y} = g$$

where the scalar force components f and g depend on gravity (and could include air resistance, although we did not) and any contact forces from the meter stick. We will discuss these forces, and the resulting criteria for flying and recapture shortly. We convert this pair of second order equations to a set of four first order equations, and we add them to the set governing the motion of the meter stick to obtain a coupled eighth order system. We integrate this system in time using the adaptive fourth order Runge-Kutta scheme given in [Press *et al.*, 1992]. We typically integrate adaptively over 100 identical subintervals so that we can accumulate simulated data at fixed, repeatable intervals of time. Our simulations use a 2 second overall interval with subintervals of 20 ms.

With this model the simulation seems to be somewhat less "whippy" than the real robot thrower. We do not have measurements of the motion of the real meter stick, so this difference cannot be quantified. Some of the difference must be attributable to the apparent extreme crudeness of the simulation model, however some is also attributable to the connection between the meter stick and the robot. Assumed rigid in the simulation, it is actually effected by bolting the stick to a rubber block that is in turn bolted to the robot. This additional degree of freedom surely contributes to larger flexion of the meter stick with respect to the robot.

To calculate a simulated throw we need to add a release criterion to the simulation. As we noted above, the experiment has no formal release mechanism: when the ball's momentum carries it out of the cup, it flies. A careful examination of frames 1-4 of the video (Fig. 3.18) suggests that the ball leaves the cup more than once, being recaptured before it can escape. This phenomenon is indeed observed in the laboratory, and is reproduced in simulation. The complete algorithm is as follows:

1. Perform a Newton-Raphson iteration on the steady state equations of motion to determine the initial "sag" of the beam with the mass of the ball attached to the stick.

2. Assume the ball is *not* in the cup and compute a 20 ms step forward in time without its mass. Also compute the motion of the ball assuming it is in free flight.

3. Check the position of the ball with respect to the cup. If the ball is determined to be in free flight goto step 5.

4. Recompute the 20 ms step forward assuming the mass of the ball is in the cup.

5. Until the ball hits the plane of the floor or 2 seconds have elapsed with the ball still in the cup goto step 2.

6. Report the position of the ball on the floor.

The first step typically converges in two iterations of the Newton-Raphson algorithm (in fact this need only be done once since its the same for every throw). The third step tests the position of the ball as shown in Fig. 3.21 (we also check for collision with the meter stick, but this never occurs in practice). This simple check handles all four state transitions:

- Ball in cup $\Rightarrow$ Ball in cup. The ball initially has the same position and velocity of the cup. If the force of gravity is holding the ball in the cup or the cup is pushing the ball, the free flight simulation will move the cup past the ball and leave it in the dashed box.

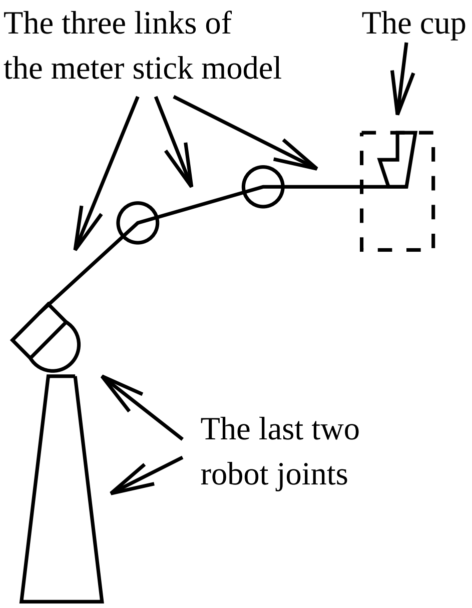

Figure 3.21: Ball release and recapture mechanism. If the ball in free flight ends within the dashed box, it is assumed to be held in the cup and not in free flight

- Ball in cup $\Rightarrow$ Free flight. If the ball's momentum will carry it away from the cup, it will not end in the dashed box and free flight will begin.
- Free flight $\Rightarrow$ Ball in cup. This is the most difficult since the capture space of the cup is hard to measure. If the free flight simulation causes the stick to pass through the ball and it ends in the dashed box it is assumed captured.
- Free flight $\Rightarrow$ Free flight. Accounted for trivially.

The landing point is determined easily by checking the coordinates of the ball as it passes through the x-y plane. If the ball is still in the cup after two seconds, we assume it will not come out as a result of the throwing motion (which is 1.2 seconds long) and terminate the throw.

3.5.2 Performance of the PUMA model

For the sequence shown in Fig. 3.18, the simulation gave a throw distance of 247 cm, compared to the actual throw of 281 cm. The final release time in the simulation is later than that in the experiment. We believe the difference in throwing distance stems from: 1. the poor modeling of the meter stick-robot attachment, 2. the crude recapture mechanism, 3. the crudeness of the three segment approximation, and (perhaps) 4. an incorrect value for the damping constant.

Some comments regarding the effects of damping and release-recapture mechanisms are in order. We looked at both effects by suppressing the natural release-recapture model described above, and calculating the maximum possible throw for a given motion by controlling the release time independently. (We calculated the ballistic trajectory for release at the end of each time step and saved the longest. Release for the longest trajectory was typically very near the end of the robot motion, about 1043 ms from the start of the 1200 ms motion.) The longest throws were computed over a range of values for the damping parameter. These maximum simulated throws are all less than the corresponding real throw, leading us to believe that the major contribution to the discrepancy between simulation and experiment lies in the modeling of the attachment of the meter stick to the robot.

The model does not appear to be accurate enough for direct learning in simulation, however we do believe the fundamental modeling problems have been handled for the flexible-meter stick ball-throwing robot. To improve the model it is at least necessary to model the compliance of the foam rubber mount and to improve the dynamics of recapture (including transmission of the ball's momentum back to the stick). An alternative is to reconstruct the throwing robot so that it can be modeled more easily. The mount could be made rigid and the release mechanism could be changed from passive to active. This would allow the release time to be included as a 13th dimension in the parameterization of a throw. Instead, we will use the existing robot and the less-than-perfect model and propose an algorithm for learning with such a model.

3.5.3 Use of the model for learning

Although, the ball thrower model as described above is not accurate enough to be directly applied to the skill learning problem, it can be used in an iterative method that improves the model while learning without requiring a prohibitively large number of real robot executions. We propose the following algorithm (it is similar to the idea of combining real and "mental" practice as seen in [Sutton, 1990], for example):

1. Perform some learning trials on the real robot using the Guided Fill In algorithm of sec. 3.2.
2. Use the results of these trials to improve the model of the robot.
3. Perform some learning trials in simulation.
4. Choose the best new trajectories discovered in simulation and execute them on the real robot.
5. Repeat.

Tests of this algorithm on the ball throwing robot look promising. We started with the 100 trial learning run on our PUMA described previously. Five throws from that set were chosen to calibrate the simulation model. The original model performed those throws with an average error of 33 cm. We use Powell's algorithm from [Press *et al.*, 1992] to minimize the average squared error as a function of the force of gravity, the mass of the ball, and the spring constant in the flexible beam. The error was minimized (at least locally) by reducing the force of gravity by 50%, reducing the weight of the ball by 20% and increasing the spring constant by 10%. These parameter settings brought the average distance error down to 10 cm. Unfortunately, changing the parameters by that much takes away their real, physical meaning. In general, it would be preferable to improve the model and retain the meaning of the parameters.

The 100 trials on the PUMA took about 45 minutes to execute. In simulation, we did 2000 trials in about 45 minutes on a SPARC 10 (note that use of GFI is still important because time still limits the number of trials we perform). From those 2000, we selected the best 25 throws and tried them on the PUMA. In that set we found several throws that went up to 8 cm farther than our previous maximum. We also found throws for medium distances that reduced the necessary control effort by as much as 40% for those distances. Finally, we noticed that throws whose motion differed significantly from any of the throws used to calibrate the model were not well predicted by the model. Executing the throws on the real robot almost always yielded a significantly shorter distance than the model predicted. Our conclusion is that the simulation can perfect throwing motions it is given, but is less likely to find qualitatively different throws that actually work on the real robot.

The inaccuracy of the model in regions of the action space where it was not calibrated is no surprise. In fact, it seems unlikely that adjustment of gravity, ball mass, and flexibility alone will produce a model accurate for all throws. There are two options to compensate for this. The first is to derive a better model of the robot arm dynamics as discussed previously. The second is to fit these parameters separately for different regions of the action space. As learning progresses the system finds appropriate parameter settings for new areas of the action space. The size of the region that can be modeled with a single parameter setting is a function of how good the original model is. If the original is almost perfect, a few settings cover the whole space and learning is quick. If the model contains almost no information, model learning deteriorates to the use of general function approximators to acquire the forward dynamics and is extremely slow.

3.6 Discussion and conclusions

There are many important tasks with highly redundant plant command to task parameter mappings. When inverting redundant mappings it is necessary to optimize according to additional cost functions. This poses a problem for standard learning methods requiring a random or brute force search of the plant command space to improve performance.

The Guided Fill In algorithm extends table lookup learning methods to redundant mappings. It uses data already stored in the table to select action space candidates that are more likely to produce good data to add to the table. Linear interpolation and extrapolation between existing good points in the table will yield more good points if the mapping is reasonably smooth. GFI also allows natural modifications to learn the easy parts of a task first since it explicitly includes a desired range of task parameters in its decision process.

Experiments with robot skill learning show that GFI can yield significant improvements in the number of training trials required to attain given levels of performance. They also demonstrate how GFI may be used to learn the easy part of a task first and the performance benefits of doing so. Many sequential task learners [Watkins, 1989] must operate closed loop because of the exponential explosion of action possibilities that occurs when a sequence of actions is considered. The results presented here demonstrate one way to deal with the large number of potential actions and thus offers an open-loop alternative for approaching these problems. The gains can be significant when the cost of perception is considered or feedback delay in a real-time system becomes a problem.

Experiments using a flexible manipulator for throwing demonstrate the power of the new learning algorithm. Researchers applying learning to robotics often attempt to constrain the action space to make the problem tractable. With efficient techniques for searching high-dimensional spaces that step may not be necessary. More importantly, the ability to handle high-dimensional spaces enables the learner to generate *qualitatively* different behaviors. Often these are the behaviors that the researcher would have eliminated by applying constraints based on poor intuition.

One of the disappointing aspects of work in learning is that it is often applied to tasks where the system designer "already knows the answer." In these situations learning functions more as a fine-tuner to improve accuracy or to fit model parameters. In the throwing experiments presented here, we had speculated that improvements could be made by storing energy in the flexible link. However, it was assumed that this would be done by making an initial backward motion, followed by a forward motion. Only through the use of the GFI algorithm was it revealed that a forward-backward-forward motion was the way to attain a high release velocity given the constraints on joint velocity, the length of time allocated for the throwing motion, and the natural frequency of the meter stick.

It is often unrealistic to assume there is no model of a skill being learned (an assumption often used in learning research). At the same time, it is only those tasks which are difficult to model that are candidates for learning solutions. We have proposed an algorithm that uses a relatively crude model to do some learning in simulation. This improves efficiency by finding good actions to try on the real system. At the same time, the execution of real actions (including those suggested by the simulation) provides training data to improve the model. This hybrid learning/modeling approach seems appropriate for real-world problems that are only possible to model approximately.

Finally, this chapter has shown the benefits of open-loop execution. Performance is not limited by the bandwidth and delay characteristics of sensors. An alternative formulation of the throwing problem might call for a closed loop execution that monitored the position of the meter stick and attempted to servo to desired release position and velocity vectors. It seems unlikely that this configuration would be capable of executing the "whipping" motion including the momentary release and recapture of the ball on long throws. Furthermore, closed loop learning requires the learning of a mapping from current states and current desired states onto actions. This kind of mapping is oriented toward short-term goals (e.g. attaining the current desired state as soon as possible as in feedback control) and often makes it difficult to consider overall task goals (e.g.

average accuracy over the whole execution as in optimal control). This difficulty will be addressed in the next chapter.

3.6.1 Applying GFI to other problems

Guided table fill in can be applied to a variety of search problems. It is most useful for inverting highly redundant mappings when data acquisition is expensive. To illustrate, consider a hypothetical problem concerning automatic design of a suspension bridge. GFI requires the definition of three things: an input space (assumed high-dimensional), an output space (assumed lower-dimensional that the input space), and a cost function (scalar). For the automatic bridge designer, the input space is a set of design parameters that can be varied continuously. The output space is a set of performance specifications (such as maximum load and expected lifetime). The cost function is the cost (in dollars) of construction and maintenance. Assume that data acquisition is expensive because the performance characteristics for any set of design parameters can be found only through expensive simulation or construction of scale models.

GFI can be applied to any problem formulated according to the above description. The result of the automatic bridge designer should be a low cost set of design parameters for a range of performance specifications. Just as the ball thrower makes use of low effort throws at short distances to find throws for long distances, the bridge designer may use cheap designs for low-performance bridges to guide its search for high-performance bridge designs. The bridge design example also illustrates how prior knowledge can be incorporated. The design parameters, performance characteristics, and cost of any existing bridges can be given initially just as sample throws were provided for the robot.

3.6.2 Future work

A possible improvement to the existing GFI algorithm is to choose specific goals rather than choosing them randomly within some range. In that way, the system might evaluate its performance over the range of task outputs and concentrate its trials on the places it is doing the poorest. As a pilot study, the algorithm was modified to try more throws at distances for which its effort value appeared to be the highest. Unfortunately, no improvement was seen in either 1-d or 2-d throwing. A potential explanation is that a learning system should spend significant time perfecting its skills in areas it is doing well so they can provide a sound basis for better overall performance improvement.

The issue of calibration is not trivial for complex systems such as the flexible thrower. The process of mounting the meter stick on the PUMA has a significant impact on its operating characteristics because of the nonrigid mount. Slight alterations in the natural frequency of the system make large differences in the results of actions. Actions that previously completely exploited the natural frequency become slightly off and throw shorter, and those that were slightly off exploit it better and throw longer. So far, no simple method has been found to calibrate previously learned actions onto future assemblies of the throwing apparatus. At present, the only reliable method has been to rerun the learning process.

A related problem is the transfer of knowledge from one skill to another. The new skill might represent small changes such as using a heavier ball or a longer flexible link. It might also be a larger change such as catching or hitting a thrown ball with the same flexible manipulator. For some cases it may be feasible to consider actions directly from the old skill as candidates for the new one. For other skills, the action spaces may not be compatible and a more complex algorithm would be required to make use of actions from the old skill. In either case "transfer of skill knowledge" remains an open and interesting problem.

4 Coaching Algorithms for Closed Loop Motor Skills

In this chapter we introduce and investigate closed loop learning by applying various algorithms to the problem of curve tracing with an inverted pendulum. This problem is well suited for demonstrating the strengths and weaknesses of learning controllers since it is nonlinear, non-minimum phase *and* unstable, thus making it more difficult than many problems attempted by learning researchers. After reviewing inverse dynamics learning and optimal control based learning we propose a method of "cooperating coaches" that we believe can address some of the shortcomings of the standard methods.

In the last chapter we discussed the advantages of open-loop motor skills. Unfortunately, there are several factors that can force us to operate a system closed loop and perform learning in a closed loop configuration. The first of these is noise. Noise is any perturbation that can enter the system and alter the state in such a way that the same control sequence from the same initial state does not always produce the same result. In all the ball throwing experiments (including those with the flexible link) our system had the desirable property that the same control sequence always produced nearly the same result. Certainly if the flexible beam acted in an unrepeatable way, it would be necessary to sense its state during the throw and modify the controls accordingly.

A second factor that can force us to run closed loop is an unmanageable increase in the dimensionality of control. This makes it difficult to learn because the space of possible actions is too large to search. Large dimensionality is often seen when the system is expected to run for long periods of time. Running for a long period of time magnifies the noise problem, but it also presents a problem in determining an initial control sequence to start. Random search for initial acceptable controls is impractical because the number of "reasonable" control sequences often becomes a smaller portion of the whole control space as the dimensionality increases. Using high level knowledge to generate some ad-hoc samples as was done with the ball throwing is more difficult for the same reason. Even with an efficient search algorithm such as GFI from Chapter 3, open-loop learning becomes impractical as the dimensionality of the control space increases.

One solution is to add some sensors and build a closed loop system. A closed loop controller provides the initial control sequences from which learning will work. It also may reduce the dimensionality of the space of controls as shown in Fig. 4.1. With open-loop control the input to the control is low-dimensional because it only contains a goal,

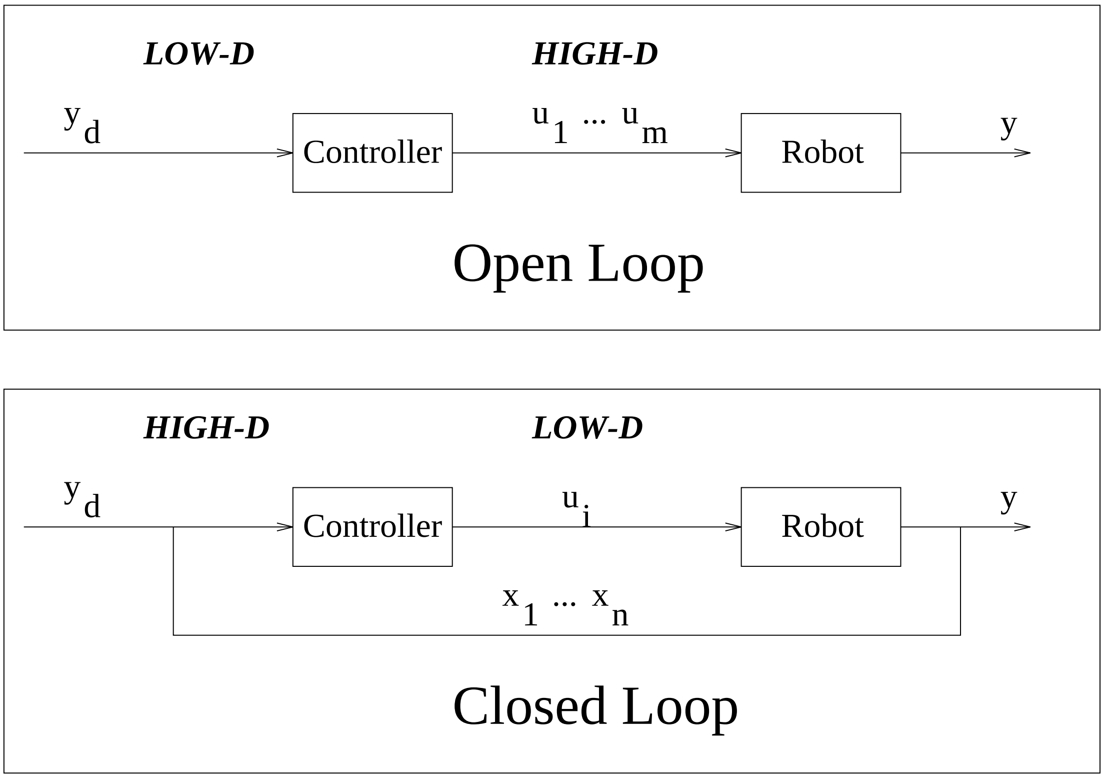

Figure 4.1: Comparison of open and closed loop control

but the output is high-dimensional because it contains the controls for each of several time steps. With closed loop control, the input becomes higher-dimensional because it includes some sensory feedback, but output becomes lower-dimensional because it has the controls for only a single time step. For tasks that take many time steps, the reduction in the size of the space can allow closed loop learning to succeed where open-loop learning would fail.

We use the architecture in Fig. 4.2 as our basic model for closed loop learning in this chapter. The box marked "controller" is always running and does not change. Its main function is to provide a reasonable control signal in portions of the state space that have not been learned yet. The box marked "learning component" outputs a control signal that is added to the one coming from the fixed controller. Its job is to learn an offset from what the fixed controller outputs that produces better results. In this chapter we look at how the learning should take place. Often, learning algorithms are considered as a whole, but we prefer to consider two separate parts called the "teacher" or "coach" and the "learner." This allows us to consider "training" or "coaching" algorithms and "learning" algorithms separately. Coaching refers to issues such as: What tasks will the system attempt during various stages of learning? How will we compute the training data from the execution of the system during learning? Learning refers to issues such as: What are the dimensions of the mapping the learner will compute? How will the learner's parameters be adjusted with each presentation of training data? Coaching

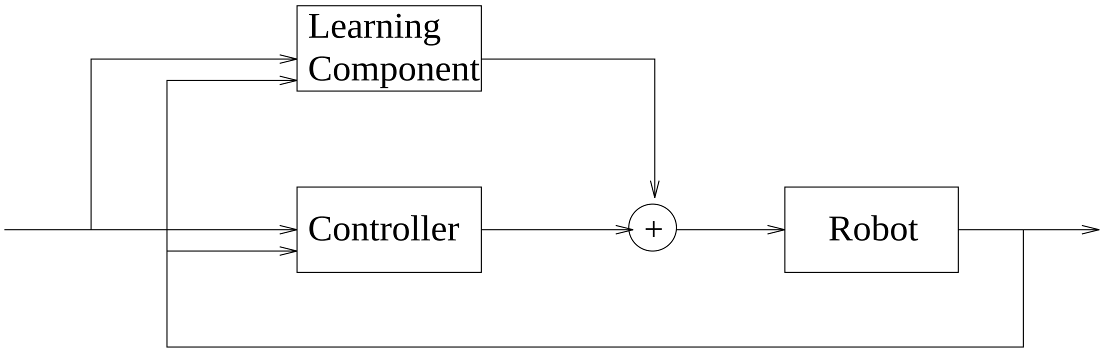

Figure 4.2: Basic closed loop learner

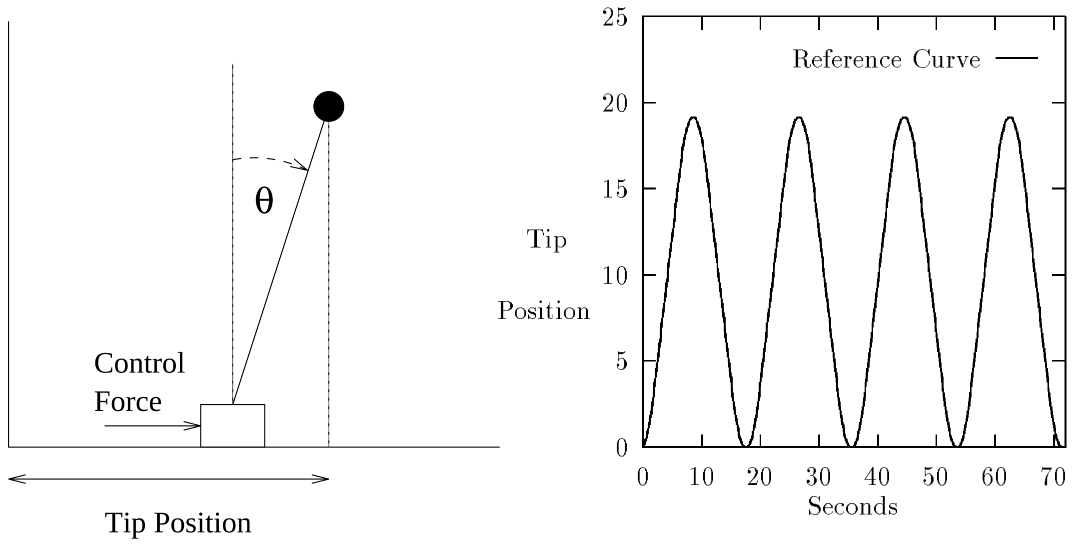

Figure 4.3: Inverted pendulum tracking task

deals with the acquisition of training data while learning deals with the processing of it.

4.1 Curve tracing with an inverted pendulum

Nonlinear, unstable plants are among the most difficult to control. As an example, consider tracing a curve with the tip of an inverted pendulum (also known as cart/pole) as shown in Fig. 4.3. The input control to this plant is a horizontal force applied at the base of the inverted pendulum (the cart). Its output is the horizontal position of the tip of the pole. The goal is to trace the curve as quickly (first term in eq. 4.1) and accurately (second term) as possible with minimal control effort (third term). There are two additional constraints on the problem: 1) There is a maximum magnitude on the control force that can be applied to the system. 2) If the pendulum ever falls to the horizontal position, the trial ends in failure. The problem is stated more formally in the notation of optimal control as:

Minimize the cost functional:

$$J = \alpha_1(t_f - t_0) + \int_{t_0}^{t_f} \alpha_2(y_d - y)^2 + \alpha_3 u^2 dt \tag{4.1}$$

subject to the constraints:

$$\begin{aligned} \dot{\vec{x}} &= f(\vec{x}, u) && (4.2)\\ y &= g(\vec{x}, u) && (4.3)\\ u_{min} \leq \ u &\leq u_{max} && (4.4)\\ \vec{x}_{min} \leq \ \vec{x} &\leq \vec{x}_{max} && (4.5) \end{aligned}$$

where the αs are relative weights, t_0 and t_f are the beginning and ending times, y_d is the desired output, y is the actual output, u is the control force applied, and $\vec{x}$ is the plant state. Note that we measure accuracy only in terms of position, but could also include velocity and higher derivatives. In practice the goal is to minimize the cost functional over a class of target curves. Here we consider a class consisting of a single target curve at different speeds.

This problem is considerably more difficult than the problem of just stabilizing an inverted pendulum which has been done with a variety of learning algorithms ([Barto *et al.*, 1983; Koza and Keane, 1990; Jordan and Jacobs, 1990] for example). The size of the state space to be learned is much larger and the desire to track curves as quickly as possible pushes the system to operate at the boundary of its instability. For the simulations in this chapter, the state equations of the inverted pendulum are the following:

x_0 - pole angle
x_1 - pole velocity
x_2 - cart position
x_3 - cart velocity
y - plant output
$\dot{x_0} = x_1$
$\dot{x_1} = c_p(g\sin(x_0) - u\cos(x_0) - c_j x_1 - c_a(x_1)^2)$
$\dot{x_2} = x_3$
$\dot{x_3} = c_c u$
$y = x_2 + c_l \sin(x_0)$

The constants are based on the length of the pole, coefficients of joint and air friction, and the mass of the cart. For this simulation, the length of the pendulum is 15 m, the mass of the cart is 1 kg, and the friction coefficients are 0.01 and 0.02. The friction coefficients were chosen such that they contribute roughly 10% of the torque on average over the reference curves we use. Note that these equations assume a system in which the mass of the pole is insignificant compared to the mass of the cart.

4.2 Instantaneous inverse dynamics

In this section we look at a fixed controller and at some learning examples based on the instantaneous inverse dynamics of the system. At each control cycle, the value of interest is the force that will make the system be exactly on the reference curve at the next control cycle.

4.2.1 A fixed controller

All of our closed loop learning algorithms start with a fixed controller that is capable of some minimal level of performance. We construct a nonlinear controller from the state equations as follows:

Using the current horizontal position and velocity of the pendulum tip, and the desired position at the next time step and assuming a constant acceleration over the interval we get a desired horizontal tip acceleration:

$$\ddot{y}_d = \frac{2((y_d - y) - \dot{y}\tau)}{\tau^2} \tag{4.6}$$

where τ is the length of the control cycle.

The horizontal acceleration of the tip is:

$$\ddot{y} = u + \frac{\partial y}{\partial x_0}\frac{\partial^2 x_0}{\partial t^2} = u + \frac{\partial y}{\partial x_0}\frac{\partial x_1}{\partial t} \tag{4.7}$$

Substituting the desired tip acceleration and the state equation for x_1 and solving for u we get the following controller:

$$u = \frac{\ddot{y}_d - 14.7\cos(\theta)\sin(\theta) + 1.5\cos(\theta)(0.02(x_1)^2 + 0.01x_1)}{1.0 - 1.5\cos^2(\theta)} \tag{4.8}$$

This controller is used with and without the terms derived from friction in the x_1 state equation depending on the goals of each experiment. It is advantageous to leave those terms out when we want to start the learner with a controller that doesn't know the correct inverse dynamics, but to leave them in when we want to demonstrate an inherent problem with a particular learning scheme.

We will compare future learning experiments against the fixed controller by itself (but without the friction terms) so we show its performance now. The reference curve is constructed with a trapezoidal velocity curve. It cycles through equal time segments of constant acceleration, constant positive velocity, constant deceleration, and constant negative velocity. It is shown on the righthand side of Fig. 4.3. Note that the integration of the trapezoidal velocity into a position reference signal looks much like a sine wave. We have three goals for tracking these curves and each is handled differently. First, we would like good positional tracking accuracy. In practice, the fixed controller always ensures that accuracy is excellent, therefore we do not discuss accuracy further.

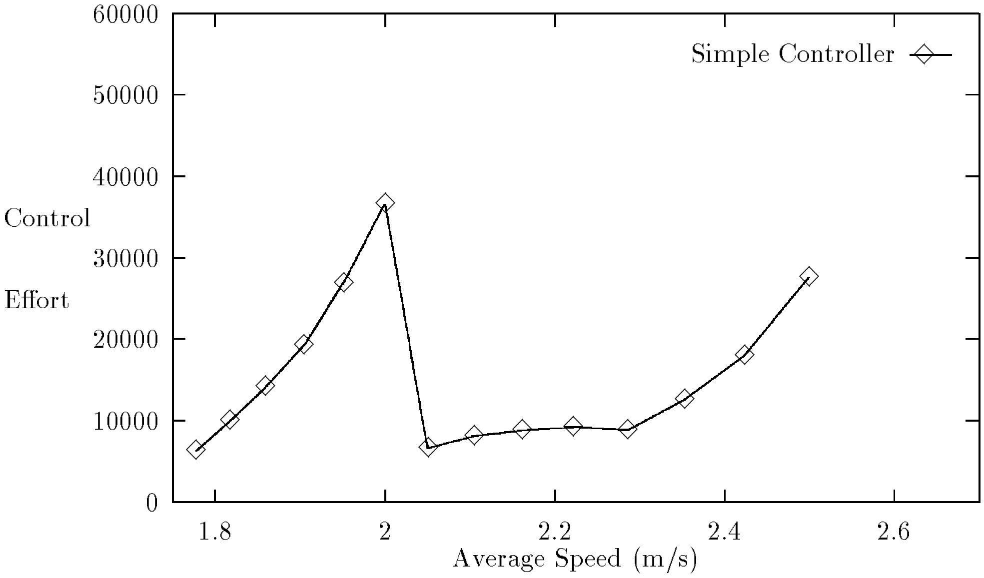

Figure 4.4: Performance of the fixed controller

Second, we would like to track the curves with the lowest possible control effort. This is measured as the sum of the squared forces as in the discrete form of the third term in eq. 4.1. This goal is addressed implicitly by the learning algorithms. Finally, we want to track the reference curves as quickly as possible. We accomplish this explicitly by presenting the system with faster reference curves. Usually slow ones are presented first and successively faster ones are given as the system learns.

The controller is discrete and chooses constant forces to apply over 200 ms intervals. The maximum force is 50 N. To see how disabling the limit is we can consider the point at which the maximum force can no longer re-stabilize the system. If the angular velocity is zero this is at angles exceeding 78^o and for a velocity of 45^o/s its at angles exceeding 60^o. Fig. 4.4 shows the performance of the fixed controller in terms of control effort and tracking speed. The vertical axis lists control effort and the horizontal axis gives the average speed the curve is traced. One goal is to lower the plot on the vertical axis and the other is to extend it farther right along the horizontal axis. The reason it stops where it does is that the fixed controller drops the pole if asked to trace the curve any faster than 2.5 m/s. Of course, the plot could go on forever to the left since the controller is able to track all slower curves easily. Note that the control effort does not monotonically decrease or increase as a function of curve speed. The interaction of the frequency of the curve being attempted with the natural frequency of the pendulum makes some curves significantly easier or harder than others of similar speed.

4.2.2 Feedback Error Learning

Feedback error learning is one example of a closed loop learning control scheme designed to learn the instantaneous inverse dynamics of a plant. It was proposed by Kawato and

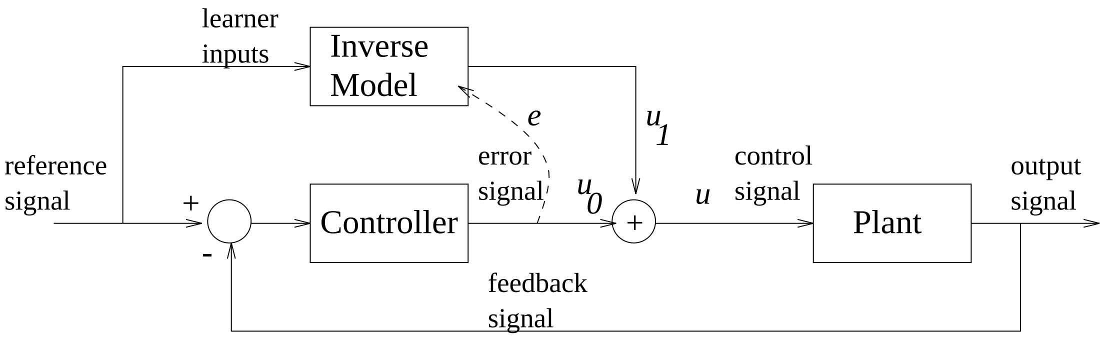

Figure 4.5: Feedback error learner

his colleagues [Kawato *et al.*, 1987; Kawato *et al.*, 1988]. A block diagram is shown in Fig. 4.5. The learning control input is the reference signal. The decision not to include any of the plant state in the feedback is motivated by the fact that parts of the human motor control system do not have quick access to proprioceptive information and thus base their output solely on the reference signal (i.e. run open-loop). Training the function approximator is done by using the control signal sent to the plant as the target signal (equivalent to saying the output of the simple controller is the error signal).

$$e = u_0 = u - u_1 = u_{desired} - u_{actual} \tag{4.9}$$

In general, the function approximator has m parameters w_i $(0 \leq i < m)$ that are adjusted to modify its output. At each control iteration, the m parameters are modified with a gradient descent update rule:

$$\Delta w_i = -\gamma \frac{\partial e}{\partial w_i} \tag{4.10}$$

where γ is referred to as the learning rate. The update rule attempts to drive the error, e, to zero. The theory is that the learner tries to reduce the error signal, thus reducing the output of the simple controller, u_0. If the output of the simple controller (computed from the tracking error, $y_d - y$) is reduced, then the tracking error must also be reduced. Perfect tracking is obtained when the function approximator has been trained such that the sum of its output and the fixed controller's output computes the inverse dynamics of the plant.

Feedback error learning has produced good results in several applications [Miyamoto *et al.*, 1988; Newton and Xu, 1993; Arai *et al.*, 1993], but there can be problems due to delays. The current feedback control signal (derived from tracking error) is due to control signals sent out in the past, therefore changes to the current output of the learner can not actually affect the current tracking error. This means that using the current feedback control signal from the simple controller as the learner's error signal yields stable control only if that signal does not vary quickly relative to delays in the loop.

Although other authors have reported success generalizing across different reference signals with feedback error learning [Miyamoto *et al.*, 1988], this may not always work.

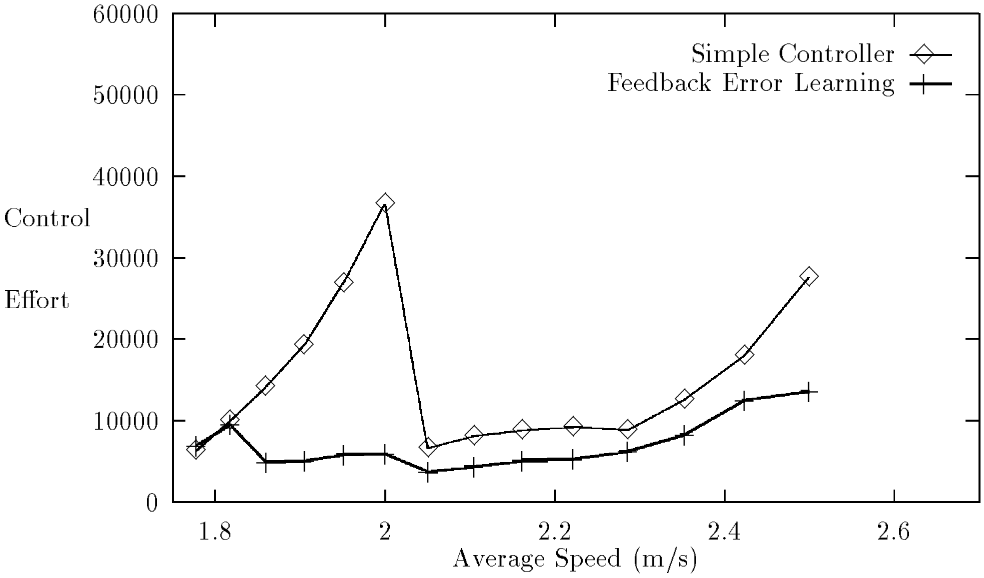

Figure 4.6: Performance of feedback error learning

The input space to the function approximator is the current reference signal, however the desired control depends heavily on the state of the plant and thus the history of the reference signal. If reference signals can vary quickly, it is easy to imagine two reference signals that are the same at some point but greatly different prior to that, thus causing the plant to be in a significantly different state. One approach to the problem is to include information about the reference signal history with the function approximator input [Newton and Xu, 1993]. For example, one might use the reference signal from the previous step *and* the current reference signal to form the function approximator input. The drawback is that the input space dimension is increased, which adds the possibility of requiring a longer training period to fill the relevant parts of the space.

Experiments with the inverted pendulum

In our experiments we use a function approximator with four input variables. They are the current and previous desired tip position and tip velocity. We start with the imperfect fixed controller that does not contain the friction terms and rely on the mechanisms of the learning scheme to adapt to this deficiency. Fig. 4.6 shows the results of using feedback error learning and compares it to the fixed controller by itself. The results are typical of what has been shown by other authors using feedback error learning. As the system learns a better approximation of the instantaneous inverse dynamics the total control effort is reduced. Unfortunately, feedback error learning has failed to do what we ultimately want: to trace the reference curve faster. It never succeeded in learning to do something that the fixed controller by itself could not do.

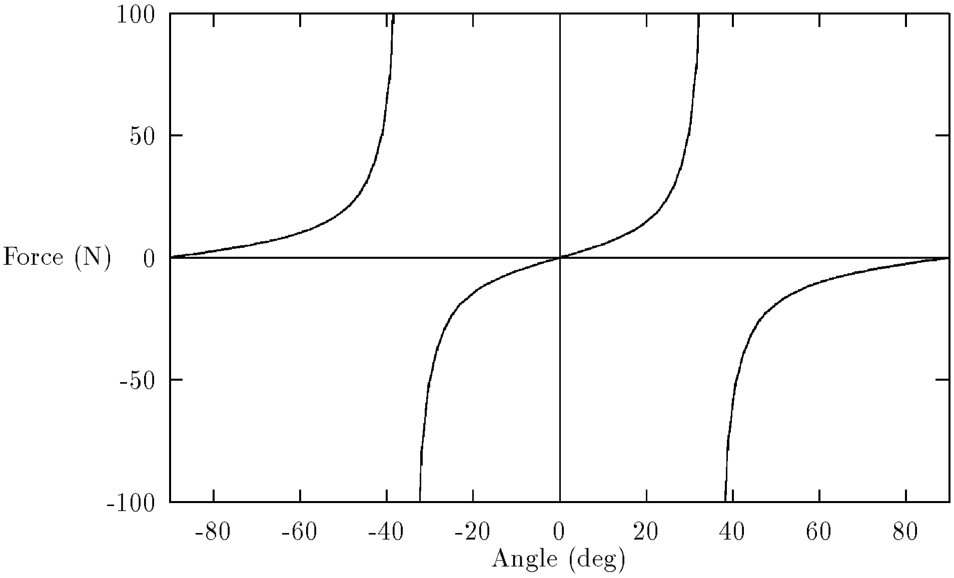

Figure 4.7: Force required to obtain zero instantaneous tip acceleration

4.2.3 Is Learning the Inverse Dynamics Enough?

Feedback error learning is just a simple scheme to learn the inverse dynamics of a system. In fact, a significant amount of closed loop learning research centers on the problem of learning the forward and/or inverse dynamics of a plant based on various assumptions about how much is already known about it. We will call these *local closed loop learners*. Once the learning is complete the controller operates by mapping the current state and desired state onto an action based on the instantaneous dynamics. There are many common plants for which reasonable performance can be obtained this way. Among them are the dynamics of most robot manipulators when operated well within the bounds of their maximum available torques. Here we will discuss when this simple method fails and show an example with the inverted pendulum.

In order to see the problems with using perfect inverse dynamics only, consider the controller of eq. 4.8. Here we will consider the controller with all the friction terms so that it is a perfect inverse model of the simulated inverted pendulum. If we assume the current angular pole velocity is zero, Fig. 4.7 shows the force required to obtain a net horizontal tip acceleration of zero as a function of the current angle of the pole. The figure and eq. 4.8 show two problems with using this controller. First, there is a singularity at 35.2^{o}. At that value the acceleration of the cart in one direction perfectly cancels out an opposite angular acceleration on the pole for any control force. The singularity will cause the controller to saturate whenever the system enters this region.

Figs. 4.8 and 4.9 demonstrate what happens when use of the inverse dynamics alone causes the system to approach the dynamic singularity. The simulated plant and controller operate in discrete 10 ms intervals (this is reduced from the 200 ms used in the learning experiments to ensure that discretization in time does not contribute to the

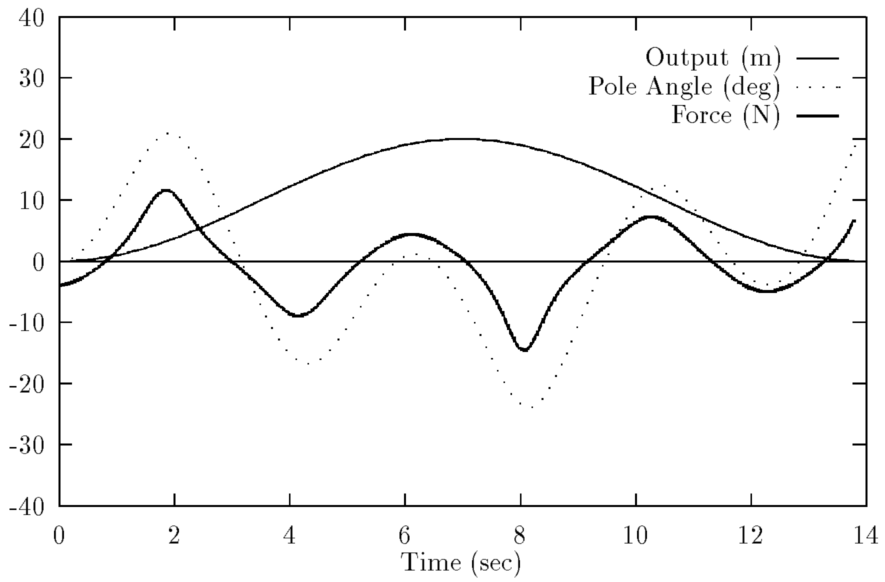

Figure 4.8: Curve tracing with perfect dynamics: 20 N maximum force, 0.071 Hz

controller problems). The reference curve is changed to a sine wave so abrupt changes in acceleration do not create control problems either. At each time step the controller computes the exact force needed to reach the goal for the next time step based on the derivation above. In each graph the reference curve is one period of a sine wave. The graphs show the position of the pole's tip in meters (the output of the plant), the angle of the pole in degrees (an indication of overall stability), and the control force used. The maximum force is 20 newtons. Fig. 4.8 shows the system having no trouble tracking a sine wave at 0.071 Hz. In Fig. 4.9 the reference curve frequency has been increased to 0.083 Hz. The graph shows that the control force saturates frequently and eventually causes the pole to fall over. The problem is that the instantaneous dynamics ignore the maximum force constraints. The controller has no way to know that the force it chooses at one instant will put it in a state requiring more than the maximum force in the future.

Consider the control for values of theta in the range $[-35.2^o{:}35.2^o]$ as shown in Fig. 4.7. If the pole is leaning left (negative angle), the control is negative. This accelerates the cart left and acts to bring the pole more upright. Similarly if the pole leans right the control accelerates the cart right and brings the pole up. A second problem with the controller of eq. 4.8 can be seen by considering the control values outside that range of theta. The desired acceleration is obtained by forcing the pole farther toward a horizontal position. The inverse dynamics are still correct, but the controller obtains the desired acceleration by moving the cart enough to compensate for angular acceleration of the pole in the wrong direction. If the pole ever tips farther than 35.2^o the controller will actively drive the pole to a crash. The controller does, however, still track the reference curve perfectly even as it is crashing the system. The problem is that the controller based on instantaneous dynamics does not use knowledge of the constraints

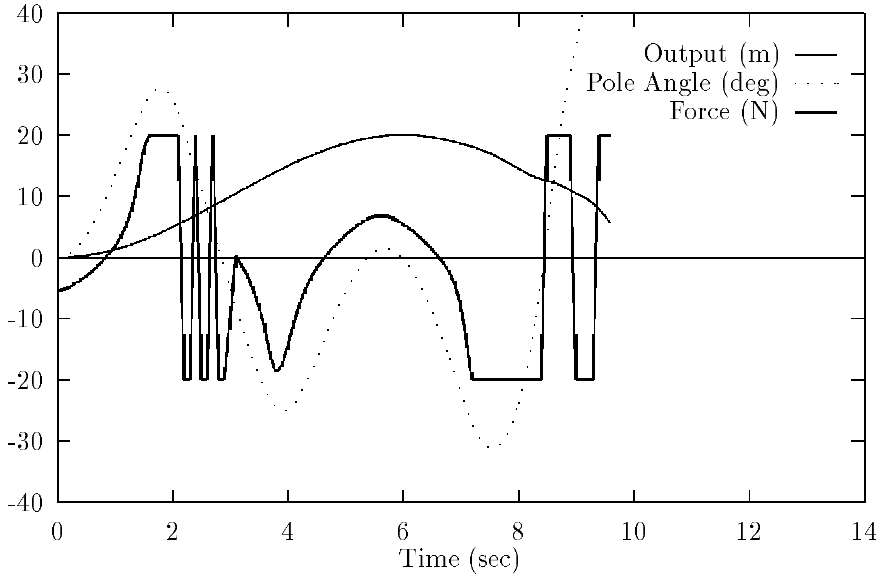

Figure 4.9: Curve tracing with perfect dynamics: 20 N maximum force, 0.083 Hz

on $\vec{x}$.

We see, then, that even if feedback error learning could do its job perfectly (learn the instantaneous inverse dynamics), it would still fail to operate the inverted pendulum at moderate speeds. The reason is that it does not consider limitations on force and constraints on state variables and thus does not prevent the system from exceeding them. What is needed is a global view of the entire task and the effects of current controls on the future state of the execution.

4.3 Optimal control

We have seen the problems with a controller based only the instantaneous effects of its controls. In order to improve the controller, it should consider the future effects of its actions [1]. This section proposes a coaching scheme that allows task level knowledge to be presented so a learning controller can use it to improve performance. This knowledge can come from several sources such as a human designer, a traditional reasoning system, or another learning controller that has already learned a certain task. We will see the benefits of this with a simple coach for the inverted pendulum. Finally, we consider a "perfect gradient" coach based strictly on the techniques of optimal control and consider how well it might perform on the inverted pendulum problem.

4.3.1 Global Coaching

[1]This is completely separate from the problem of control in the presence of delay which can also force a controller to predict and consider the future effects of its actions.

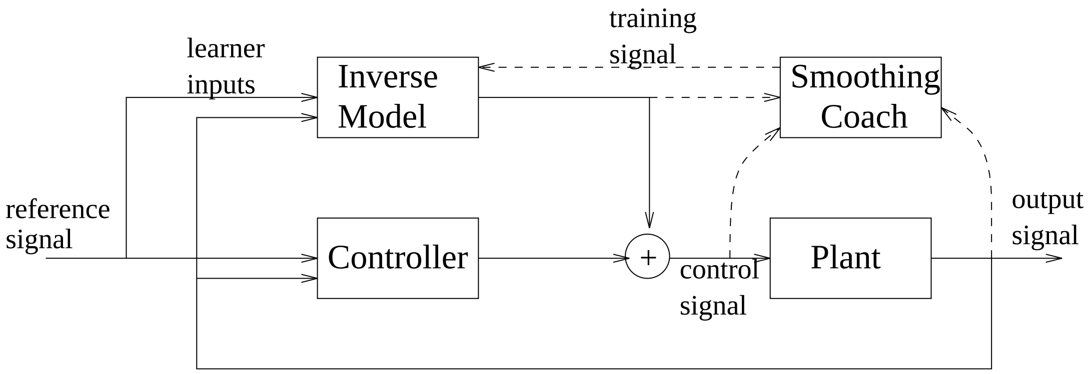

Figure 4.10: Learning with a smoothing coach

Fig. 4.10 shows a modified learning controller (proposed in [Schneider and Brown, 1994]) that addresses the potential problems with basic feedback error learning. Others have shown the benefits of adding information about earlier portions of the reference signal along with the current values [Newton and Xu, 1993]. Doing so gives information about the current state of the plant by considering where it should have been previously. Refusing to look at the plant state is motivated by biological considerations [Kawato *et al.*, 1988], but it is probably not the best way to produce a learning controller when that state information is readily available. Therefore, we use plant state in the input space of the function approximator. It captures the relevant part of the execution history and should improve the generalization capability.

There is a second problem shared with all local closed loop learners. The update rule for feedback error learning (eq. 4.10) attempts to reduce the second term of the cost metric (eq. 4.1). However, it only directly reduces the error at the current step. The effects of the modification may *increase* the tracking error in the future by putting the plant in a less desirable state. Under some assumptions about the plant it is possible to compute the gradient of the *total* cost metric with respect to the parameters of the function approximator, w_i (we discuss this later). Here, we address the case where this is not possible and propose a method that heuristically takes a global view of the cost metric and attempts to reduce it.

Rather than exclusively training on-line during execution, we record a trace of the control signal, plant state, output, and reference signal over an entire execution $(u(t), \vec{x}(t), y(t), y_d(t))$. In our experiments these are all discrete rather than continuous functions. The trace can be viewed and altered by a system designer, teacher, or "coach," or by an algorithm incorporating appropriate heuristics. The modification yields a new control signal, $u'(t)$, to be used for training rather than the $u_{desired}$ of eq. 4.9. Learning is done as before (except it is off-line) using the new desired control signal. This method has potential as a new way to approach controller design, since it allows arbitrarily general or specific guidance.

With this coaching scheme we hope to obtain some of the characteristics of human skill learning. For example, a coach might suggest that you "bend your knees more while skating backwards." Doing so will directly and indirectly affect the sequence of system states you go through, but eventually the advice gets encoded into the control system and you reap the performance benefits. One can imagine a system designer creating an improved controller with the same kinds of suggestions, or "rules of thumb." Converting high level rules into controllers is often approached with fuzzy logic ([Handelman *et al.*, 1991] for example). Whether these systems suffer the same problems as local closed loop learners usually depends on the complexity of the rules given by the system designer.

Ridding a control system of an incorrect behavior is an interesting problem that has been encountered by other learning systems (ALVINN [Pomerleau, 1989] for example). The use of a high level coach with off-line learning addresses this issue also. At the end of an execution, the performance may be evaluated. If it is unsatisfactory the trace can be discarded completely without doing any learning. On-line closed loop control learners never have the opportunity to make this decision because they do not have the global performance available during execution.

Experiments with a smoothing coach

We constructed a coach for the inverted pendulum tracking problem. In these experiments, the input space of the function approximator is three-dimensional. The dimensions are: the current angular position of the pole, the current angular velocity of the pole, and the desired tip acceleration as given in eq. 4.6.

Observation of the system in operation shows that it usually drops the inverted pendulum as the result of wild oscillations in its control signal. These wild oscillations also use high accelerations and thus increase the control effort. Therefore, we chose to use a coach that modifies the control signal by passing a smoothing filter over it:

$$u'_t = \sum_{i=-s}^{i \leq s} a_i u_{t+i}, \quad t_0 < t < t_f \tag{4.11}$$

We use $s = 1$ and $a_i = 1/3$ for all i. Fig. 4.11 shows the results of using this smoothing coach. The control effort for some speeds is higher than with plain feedback error learning, but the system is now able to trace the trajectory up to a speed of 2.7 m/s compared to a best of 2.5 m/s in the previous trials. However, the coached controller causes the system to drop the pendulum at a speed of 1.9 m/s even though the fixed controller alone is capable of performing at that speed. Two things can be concluded from observing these experiments. First, the smoothing coach worked to increase stability and allow the system to track the reference curve faster. Second, a large acceleration is still needed to reverse the direction of motion during tracking and the coach acted to inhibit that acceleration. That inhibition resulted in higher control effort and instability at some speeds.

4.3.2 Perfect gradient coaching

We have seen that it is possible to get improved performance from a coach we construct by considering the entire task being attempted. In this section we consider a "perfect

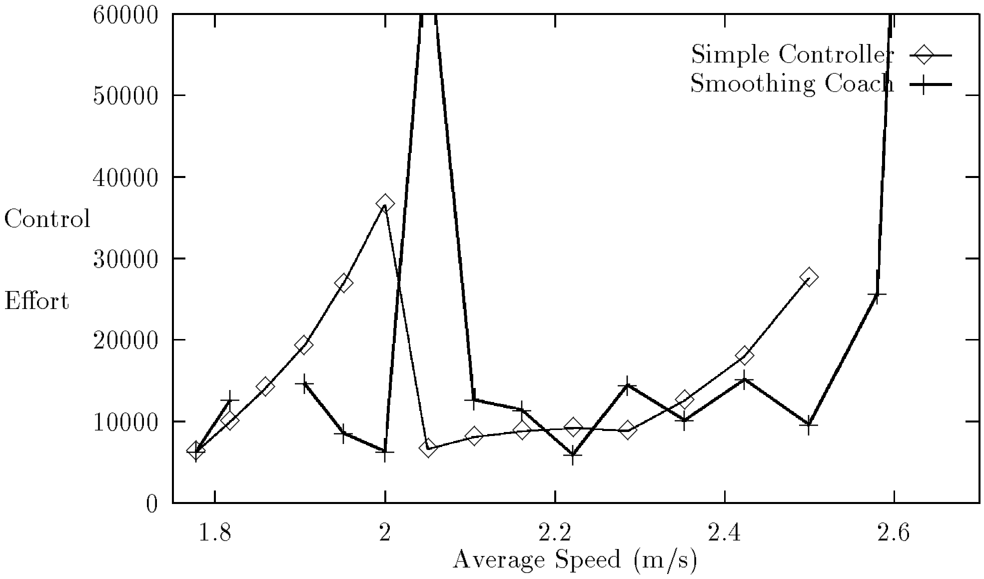

Figure 4.11: Performance of smoothing coach

gradient" coach. A perfect gradient coach is one that suggests a control to optimize the overall performance of the system on the remainder of the task. Finding a controller to produce the true optimal control signal with respect to minimization of a cost equation such as eq 4.1 is the subject of optimal control theory (see [Lewis, 1986] for an overview). For our inverted pendulum problem that means choosing a control at each time step that results in a minimum value of the cost function within the constraints provided by eqs. 4.2 through 4.5.

In practice, few optimal control problems can be solved in closed form and the inverted pendulum tracking problem is not one of them. A common alternative is to use numeric solutions that depend on a form of gradient descent. There are many variations, but all choose an initial control trajectory and then enter a cycle of: execute control trajectory, compute gradients of cost with respect to control for the states the plant went through, modify the control trajectory based on those gradients, repeat. We define a perfect gradient coach to be one that has perfect access to this gradient information and can thus suggest updates to the controller that will move its performance toward the minima in the cost function. The question we address next is: *If we could find such a coach, would we be guaranteed a solution to a problem?*

Consider the following simple optimal control problem with the inverted pendulum:

- The pendulum begins with the cart stationary at 0, the pole at an angle of 32^o and an angular velocity of -20^o/sec.
- The system operates for two time steps with each step lasting 1 second.
- The reference trajectory is [6.5, 9.0] (the initial tip position is 7.95 m).

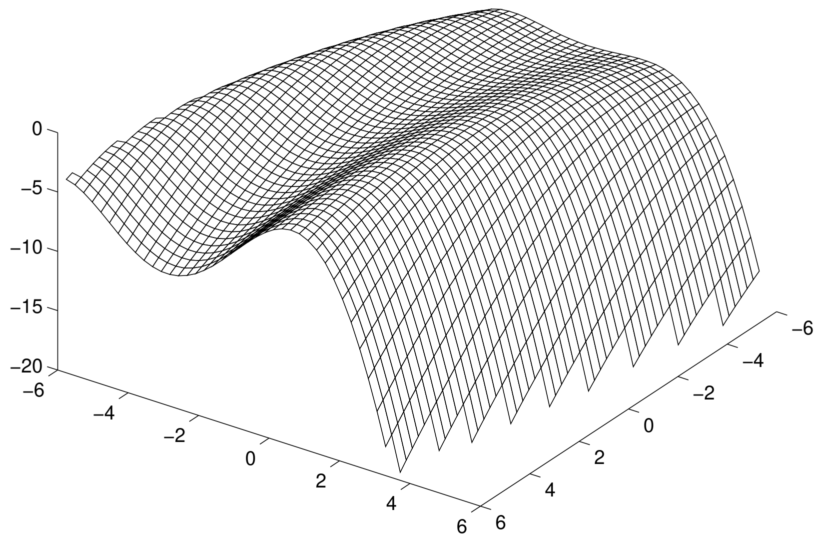

Figure 4.12: Mesh plot of (negative) cost as a function of the two forces

- The cost function is $(tip_1 - 6.5)^2 + (tip_2 - 9.0)^2 + 0.05((force_1)^2 + (force_2)^2)$ where tip_i is the horizontal tip position at time i and $force_i$ is the force used at time i.

In words, the problem is to find a control force that will move the tip from 7.95 to 6.5 during the first time step, and a second force that will move the tip from there to 9.0 during the second step. Additionally, the first force should be chosen to set the system up for easy accomplishment of the goal at the end of step 2 (i.e. the best solution may trade off accuracy and effort on the first step to realize lower cost on the second). Figs. 4.12 and 4.13 show the cost as a function of the two control forces. The negative of cost is plotted to improve the visualization. The plots show two local minima in the cost function. There are two ways to track the given reference curve. One (shown at the top of Fig. 4.14) is to apply an initial positive force to stabilize the pole and then bring it to the goal point at step 2 with almost no force (the local minima with this method uses forces of 1.35 and 0.61 to get a total cost of 0.28). The other (shown at the bottom of Fig. 4.14) is to apply a negative force that tips the pole farther, but reaches the goal points while doing so (the local minima with this method uses forces of -3.31 and -1.35 to get a total cost of 0.70). In this example, a perfect gradient coach is not enough to ensure the best control policy. The final policy is determined by which basin of attraction the initial control trajectory falls in and the local minimum solution of one basin has higher cost and worse stability than the other. This illustrative example demonstrates a large obstacle for closed loop learners. Even if the learner can form a perfect model of the task and the plant and use an optimal control formulation of the problem to compute gradients for the controller parameters, it may still fall victim to local minima. Virtually all robot learning algorithms that rely on gradient descent depend on the local minima being acceptable.

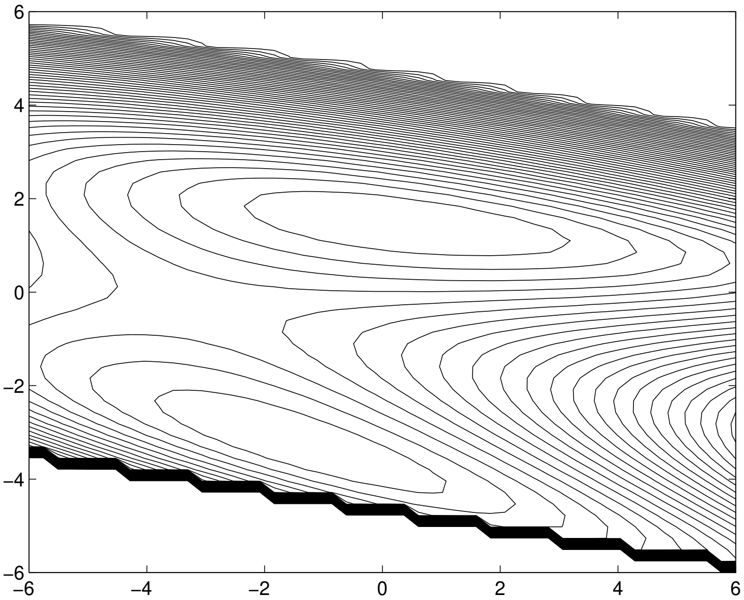

Figure 4.13: Contour plot of cost as a function of the two forces

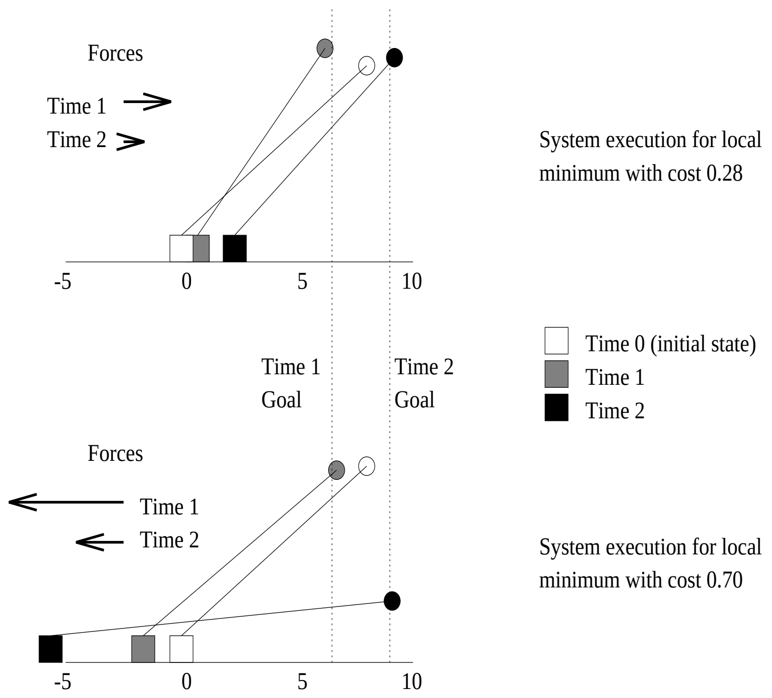

Figure 4.14: Traces of system execution for each of the local minima

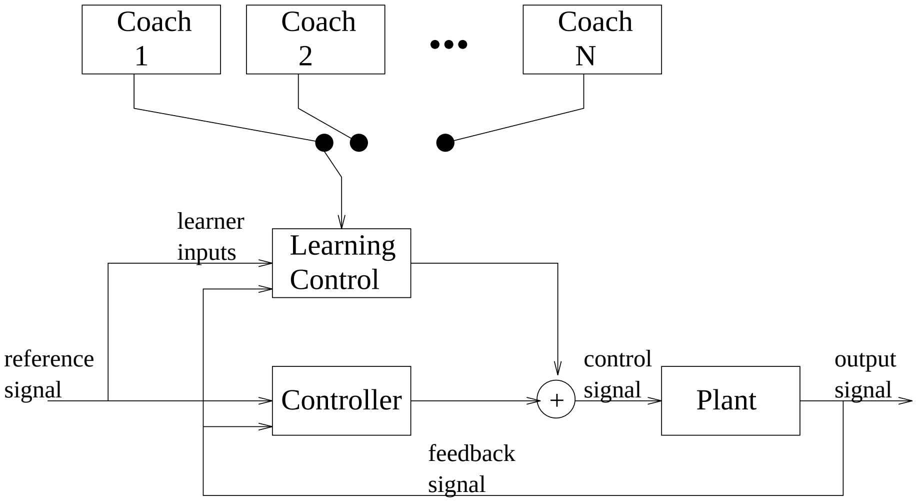

Figure 4.15: Cooperating coaches

4.4 Cooperative learning

In this chapter we have seen how heuristic information about a task can be used to improve system performance with a closed loop learning controller. We have also seen that even a perfect gradient coach may not be able to get the system to its best possible performance. With these two points in mind, we propose an architecture of *cooperating coaches*. First we look at a simple model that uses multiple coaches to guide the search through a learning controller's parameter space. Then we will generalize the model to allow several learning controllers to work together and we discuss the benefits of their doing so. Finally, we will see how these architectures perform on the inverted pendulum.

4.4.1 Cooperating Coaches

Fig. 4.15 is a block diagram of the architecture for cooperating coaches that work together to learn a particular task. During each trial, only one coach is active. It provides the training signal to the learning controller. At the beginning of each trial a decision is made about which coach will be active for the trial. Usually, this is best done by considering the performance of the previous few trials.

Cooperating coaches is an idea intended to overcome the local minima problems that may occur with optimal control gradient descent learning control solutions. We illustrate the idea with a simple example of function minimization. Consider the function at the bottom of Fig. 4.16. It is the sum of the three sine waves plotted above it. Suppose we use gradient descent and begin at 4.0 on the x axis. The process will quickly stop at a local minimum. Now suppose we start with gradient information from the function at the top. From there we reach a local minimum around 3.5. Then suppose we switch to gradient information from the second function to get to 2.7 and to the third function to get to 1.4. From there we can use the gradient of the original function to get to 0.9.

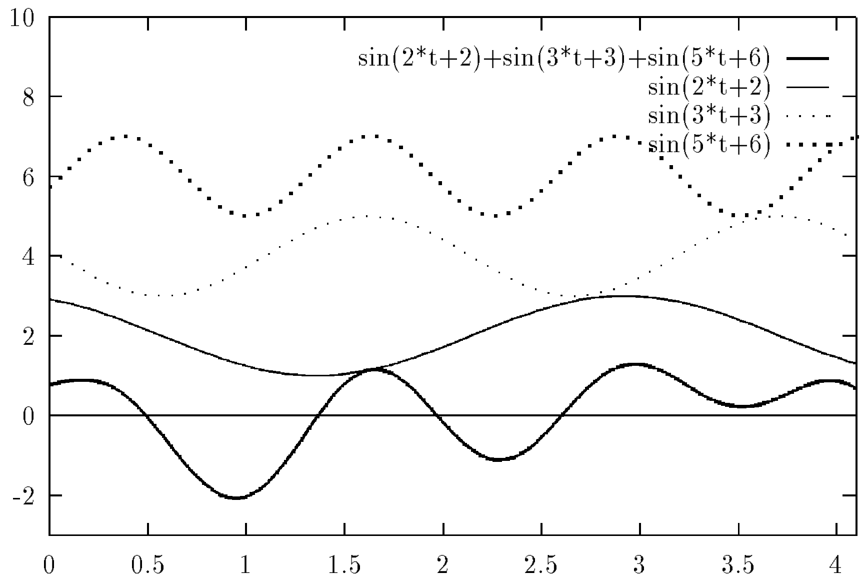

Figure 4.16: A function minimization example

Of course this is not a particularly useful way to do function minimization in general. For skills, however, each of the component functions represents one of the coaches that makes use of some heuristic information to guide the search in the right direction. Each of the coaches individually can suffer from local minima or even instability of the learning process, but by switching between them a better solution may be found.

4.4.2 Generalization to Cooperating Learning Controllers

Fig. 4.17 is a block diagram for a generalized version of cooperating coaches we will call cooperating learning controllers. This architecture removes many of the restrictions present in the cooperating coaches model. Cooperating coaches uses a single function approximator operated on by several different coaches. This means that the system is always at only one point in the learning controller parameter space. With cooperating learning controllers, each coach is grouped with its own learning controller. This allows parallel consideration of several points in the learning controller parameter space since the function approximator in each learning controller may represent different points in the space.

Cooperation occurs implicitly in the cooperating coaches model as each coach takes its turn manipulating the learning controller, but it is explicit in the cooperating learning controllers model. Again, only one coach is active during each trial (as indicated by the switch), but any other learning controller may watch the active coach and use its output as a training signal for itself. Under this scheme each coach may attempt to track a new, difficult curve in its own way when its turn comes up without doing damage to the whole system. If it fails, none of the other learning controllers will train on it and its errors will not be incorporated into their function approximators. However, when

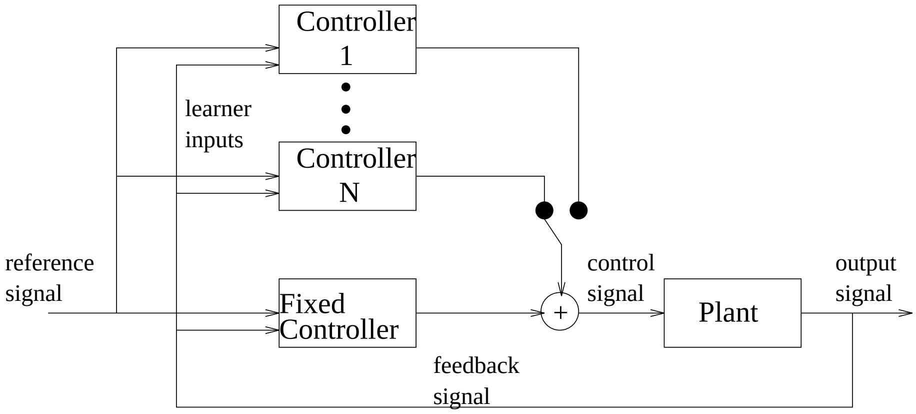

Figure 4.17: Generalization to cooperating learning controllers

one controller succeeds, any of the others may learn from its success. The selection algorithm now must decide which coach to activate *and* which others will watch and learn from it.

This generalization allows additional flexibility. There is only one set of learning controller inputs for cooperating coaches since there is only one function approximator. In cooperating learning controllers, however, each one can have a different set of inputs. One may use only the reference signal and run open-loop as in traditional feedback error learning, while another makes full use of the plant state. Some parts of a task may be easier to learn with different inputs than others.

When the learning is complete, cooperating coaches need only turn off the training from the coaches. With cooperating learning controllers there is a remaining issue about which should execute during normal operation. One possibility is to select the one with the best overall performance. Another is to select different controllers dynamically during execution for different types of reference curves. This is similar to the "mixture of experts" idea in classification [Jacobs *et al.*, 1991] or the combination of control laws in fuzzy control.

4.4.3 Experimental Results with Cooperation

In the experiments described above, feedback error learning does better against some performance metrics (lower control effort) while the smoothing coach controller does better against others (faster tracking speed). In the following experiments these two systems are combined as cooperating coaches and as cooperating learning controllers. Each coach considers the same data as it had previously. We use the following heuristic algorithm to control the selection of coaches.

1. Increase the desired tracking speed.

2. Run trials with the smoothing coach until the trajectory can be tracked stably.

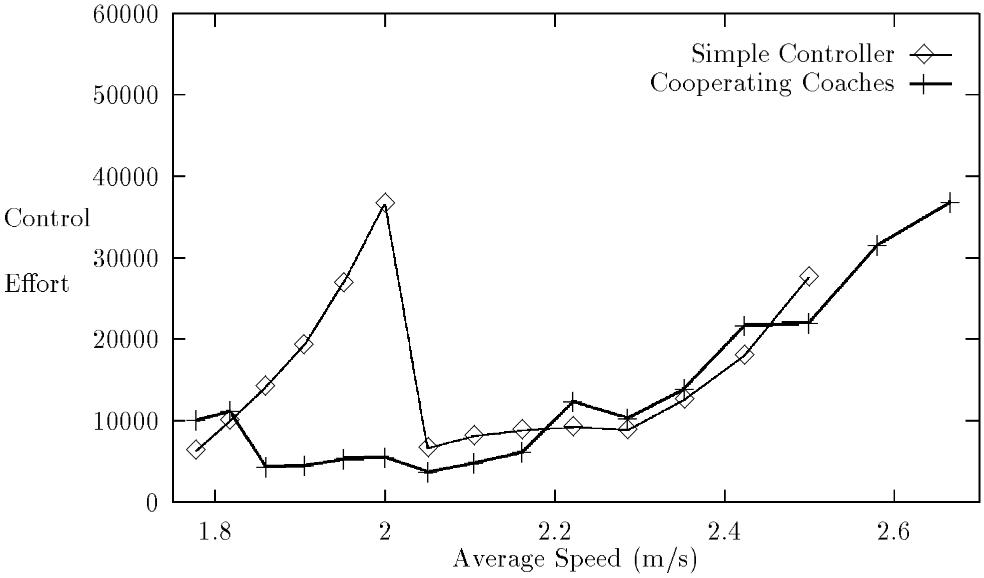

Figure 4.18: Performance of cooperating coaches control

3. *Cooperating learning controllers only:* Train the feedback error learning controller on the stable execution by the smoothing coach.

4. Switch over to the feedback error learning coach.

5. Run trials until no further improvement in control effort is observed

6. *Cooperating learning controllers only:* Train the smoothing coach controller on any feedback error learning execution that lowers control effort.

7. Repeat until step 2 fails to track a faster trajectory stably.

8. *Cooperating learning controllers only:* Final executions are done with the feedback error learning controller.

Experimental results using cooperating coaches are shown in Fig. 4.18. The combined system tracks trajectories as fast as 2.7 m/s, just as the smoothing coach does when operated alone, while correcting the problem the smoothing coach had with the 1.9 m/s trajectory. It also uses 55.1% less control effort than the smoothing coach does when operated alone. It does, however, use 40.0% more control effort than feedback error learning at the speeds feedback error learning is capable of doing by itself. The resulting system successfully combines the strengths of the two controllers to produce a superior result.

Results of experiments using the cooperating learning controllers architecture are shown in Fig. 4.19. This architecture is also capable of tracking trajectories up to 2.7 m/s. It uses 60.7% less control effort than the smoothing coach alone and 19.0% more than feedback error learning on the trajectories it can track. These numbers are

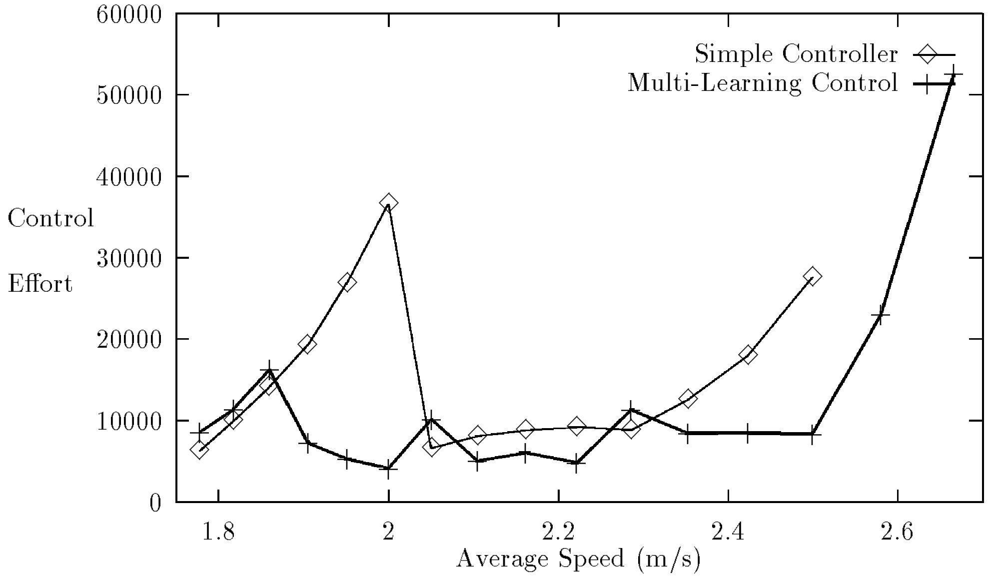

Figure 4.19: Performance of multi-learning control

better than the cooperating coaches model, but not enough to make a strong statement about which model is preferable. The cooperating learning controllers model is at least as powerful as cooperating coaches, because cooperating coaches is just a special case where each controller uses the same parameters in its function approximation and always learns from the active coach. However, the cooperating coaches model may be preferable based on its simplicity.

4.5 Conclusions and open problems

We have considered closed loop learning controllers using the problem of trajectory tracking with the tip of an inverted pendulum. This problem captures the important aspects of motor skills: difficult job, unstable plant, with premium placed on accuracy, efficiency, and speed of performance. Many closed loop learning controllers, including feedback error learning, use only the current state and error to compute a training signal for the learning algorithm. This method is simple, efficient and can be useful when applied to stable plants that do not have severe limits on their controls. For more difficult plants, we have proposed a "coaching" scheme that allows high level heuristic information to be incorporated into a learning controller. In our experiments, a smoothing coach was able to track faster trajectories than plain feedback error learning. A remaining open problem is how to produce appropriate coaching heuristics and how to evaluate and predict their performance for various classes of plants.

The new method of generating a training signal opens the possibility of a new way to design controllers for complex, nonlinear plants. It captures the concept of a coach observing a student and suggesting improvements to his form. A system could provide a designer with an image of a control signal after each training execution. The designer could then use an editor to correct what he perceives as "mistakes" in the process. This might be particularly useful in systems where "mistakes" result in catastrophic loss of stability.

Finally, we have proposed a method to combine learning controllers with different properties to obtain a new one with the best properties of each. This method addresses the problem of local minima in the learning process by switching between coaches. One coach actively trains the system at each execution, but control may be switched between several coaches during the training. In our experiments, two controllers were combined to produce one that accomplishes faster trajectory following and reduced control effort. Although our experiments used only two controllers, it is possible to use any number. The controller's coaches might differ in the type of heuristic, the parameters of a particular filter, or function approximation techniques. Each learning controller could "get its shot" at reference curves from the training set and the first one to succeed would try to pass "the answer" on to the others, as in other forms of distributed search (e.g. [Bianchini and Brown, 1992]).

An important open problem is to find a suitable selection algorithm that chooses the active controller during learning. We use simple algorithms in our experiments, but it is likely that the selection algorithm is crucial for good performance. One possibility is to label each coach as good for improving some aspect of the performance metric. When a reference curve is presented, the selection algorithm could decide which aspect of the metric needs to be improved most for that particular curve and choose the coach most likely to do so. A more sophisticated algorithm would learn which coaches are best at making which improvements based on their performance on previous curves.

There is a danger that training by a series of different cooperating controllers might cause interference and result in a non-convergent process. At the present we have not investigated the conditions when this might occur. We have explicitly chosen to develop algorithms for maximizing performance in cases where typical assumptions about plants and reference signals do not hold. However, we view the further mathematical analysis of these algorithms as an interesting and important problem.

5 Closed Loop Learning as an Open Loop Search Technique

In Chapter 3 we discussed techniques for efficient open-loop skill learning. We observed that open-loop execution, and thus open-loop learning, have the advantage that bandwidth is not limited by the characteristics of the sensors, and that feedback is not present to put limits on gain, affect stability, and compromise performance. In Chapter 4 we saw that closed loop execution, and thus closed loop learning, is necessary sometimes and discussed techniques for doing so efficiently. In a complex system performing numerous open and closed loop tasks, there is another reason to prefer open-loop execution. There may be limited sensing resources in the system. In the presence of these limits the ability to perform certain tasks open-loop, or with reduced sensing, can allow the system to increase the number or complexity of tasks it performs simultaneously.

In this chapter we assume the existence of a successful closed loop learning system and investigate how the need for sensor information can be reduced as the system becomes more skillful at performing its tasks. This reduction comes when the system decides not to use its sensors during some time steps. As learning continues it should choose not to use them for longer periods of time. A system that successfully eliminates its need for sensing has used closed loop learning as an efficient search technique to acquire an open-loop skill.

In Chapter 4 we discussed the use of feedback error learning as a closed loop learner. In its original form the function approximator does not accept sensor information about the current robot state to compute a control signal. Its output is only a function of the reference signal. One of the reasons for this is so the system can eventually perform open loop. As the learner improves its mapping, the output of the fixed controller, which uses sensor information, is reduced. Ideally, its contribution to the system would converge to zero and it and the sensors could be shut off. Unfortunately, this does not happen in practice. The output of the fixed controller is generally reduced, but noise and inaccuracies from the learner make it impossible to remove it and the sensors. Rather than reducing the contribution of a fixed controller that uses sensed information, the scheme we describe in this chapter minimizes dependence on sensing by substituting estimated for sensed information over increasingly long periods of time. Even if it fails to operate completely open loop, it still frees some sensing resources. The advantage can be translated into maintaining performance with less sensing or into using the same amount of sensing to increase performance. Quantifying the reduction in necessary sensing is another way to measure the gains provided by learning.

5.1 Closed-to-open loop architecture

Fig. 5.1 is a block diagram of the architecture. There are three separate function approximations. Their mappings are:

F1: $u = F1(\vec{x}_f, \vec{x}_d)$

F2: $\vec{x}_{est} = F2(\vec{x}_f, u)$

F3: $p = F3(\vec{x}_f, u)$

where $\vec{x}$ is a vector of current states, $\vec{x}_d$ is a vector of desired, or reference, states, $\vec{x}_{est}$ is an estimated state vector, $\vec{x}_f$ is the feedback signal used by the controllers, u is a control signal, and p is a confidence value between 0 and 1 as indicated in Fig. 5.1. The learning rules for these function approximators are presented separately below. First, we overview the system by describing one control iteration. A reference signal is presented at the input on the left. Plant state information is available with the reference signal. As in the learning controllers of Chapter 4, there is a fixed and a learning controller (F1) that compute control values to be summed. The control is fed to the plant. In parallel with the plant is a plant model (F2) that accepts the current state and control and computes an estimated state. Feedback is provided through sensors or the estimated state. A confidence predictor (F3) determines how reliable the estimated state is and decides whether to use it or to activate the sensors.

In the experiments below, as in Chapter 4, each of the function approximators is implemented with a form of radial basis functions using non-movable centers. Each one has its own learning rate, α, which may be constant or variable during execution. As the system learns, improvement in F1 causes the system to perform the task more accurately, while improvement in F2 reduces the need to use sensing.

5.1.1 F1: learned controller

The explicit purpose of F1 is to learn a good controller for the given task. It also has a secondary purpose, which is to guide the experimentation that provides training data for F2. A random training method would call for F2 to learn by providing the plant with a sequence of random control signals and thus force it to learn the mapping for the entire space. However, it is more efficient to learn the plant mapping only in the portion of the state space needed to perform the desired tasks. In this architecture F1 and F2 are trained simultaneously with F1 choosing controls (and thus training data for F2) that are expected to perform the current task well.

At the end of each control cycle, we would like to compute a new training data point for F1. We already have the input values $(\vec{x}, \vec{x}_d)$ and need to compute a desired control signal, u_d. There are two other values that can be used to compute it: the control used

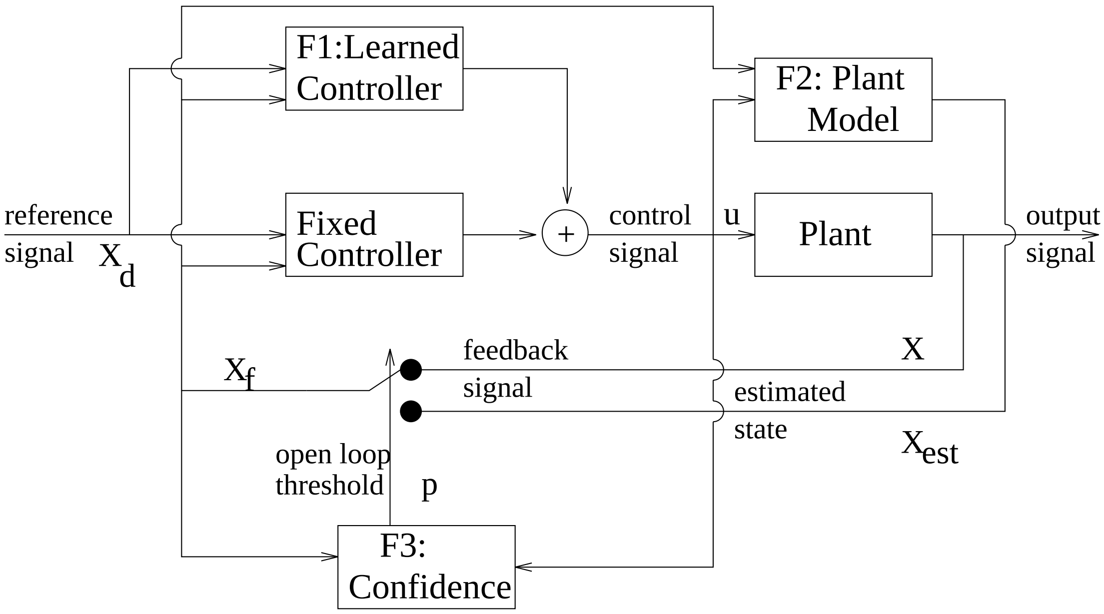

Figure 5.1: Closed-to-open loop architecture

and the cost incurred. If we assume the ability to compute the derivative of J with respect to u and have a desired change in the value of J we can compute u_d as follows:

$$(\vec{x}, \vec{x_d}) \rightarrow u_d = u + \frac{\Delta J_d}{\frac{\partial J}{\partial u}} \tag{5.1}$$

where ΔJ_d is the desired change in the cost. If tracking error is the only cost, ΔJ_d can be set to the negative cost of the current step.

5.1.2 F2: plant model

The purpose of F2 is to provide a model of the plant that can be substituted for real sensing of the plant state. Training data is of the form: $(\vec{x}_t, u_t \rightarrow \vec{x}_{t+1})$. One way to get this information is from a sequence of: {sense state, send control, sense state}. We may also be able to use a sequence of: {estimate state, send control, sense state}. Note that the sequences: {sense, control, estimate} and {estimate, control, estimate} are not useful as training data because they amount to training the function approximator on itself.

5.1.3 F3: confidence in the plant model

The value p, as computed by F3, is the confidence in the estimate provided by F2 when given the same arguments as F3. A value of 1 indicates that the value computed by F2 for those arguments is definitely correct, while a value of 0 indicates that the value computed by F2 contains no information about the real value of the mapping at that point. For a given tuple of F2 training data, $(\vec{x}_t, u) \to \vec{x}_{t+1}$, we compute the p value needed for F3 training data as follows:

$$\vec{x}_{diff} = \vec{x}_{t+1} - F2(\vec{x}_t, u) \tag{5.2}$$

$$p = e^{-\|\vec{x}_{diff}\|^2} \tag{5.3}$$

First we determine the difference between the desired output in the training data and the current output of F2 on the input of the training data. If this is small we are fairly confident about the approximation of F2 in that region of the state space. If the difference is large, the approximation has not yet converged or there is considerable noise in the system and we are less confident about it. We convert the vector into a confidence value with a gaussian function.

F3 determines a confidence in the current estimated state given that the arguments provided to F2 are correct. If the system runs open-loop for several steps these arguments will only be estimates. Therefore, a running product of the confidence values provided by F3 is kept as long as the system runs without sensing. When the running product falls below a threshold (referred to as the *open-loop threshold*), the switch is set to make a real sensor reading and the product is reset according to the confidence in sensing (1.0 if the sensors are perfect).

5.1.4 Extending the learning rules over several steps

The previous sections describe the learning rules for each function approximator at a single step where sensing was done (called the 1-step learning rules). Ideally, training data could also be generated from the steps where there was no sensing. Assume there have been $n-1$ steps without sensing followed by one with sensing. There has been an accumulated cost of J and we have a desired cost reduction during those steps of ΔJ_d. Define the estimation error as: $\vec{E}_e = \vec{x}_{est} - \vec{x}$

The blame for the cost and estimation error must be distributed over the last n steps even though there is no information except the sensing on the last step. In the absence of other information, we assume each step contributed equally to the errors which leads to the following rules for producing training data (called the n-step learning rules):

F1: $(\vec{x}, \vec{x_d}) \to u + \frac{\frac{\Delta J_d}{n}}{\frac{\partial J}{\partial u}}$

F2: $(\vec{x}_t, u_t) \to \vec{x}_{est,t+1} + \frac{\vec{E}_e}{n}$

F3: $(\vec{x}_t, u_t) \to (e^{-\|\vec{E}_e\|^2})^{\frac{1}{n}}$

The first two rules are obvious extensions to the 1-step learning rules. The reasoning behind the rule for F3 is as follows: Use the gaussian from before to convert $\vec{E}_e$ into a confidence value. Then observe that we would like to arrive at that confidence value at

	Going from Closed to Open Loop	State Estimation (Kalman Filter)
Goal:	Reduce use of sensors	Determine plant state
Assumptions:	Sensors are accurate, but costly Plant is noisy	Sensors are noisy Plant is noisy
Method:	Use sensors to create plant model Replace sensing with plant model	Assume plant model Use sensors to overcome plant noise Use plant model to overcome sensor noise
Decision at each step:	Can we use the estimate and do without sensing?	What relative weighting should be given to the sensor data and previous estimate?
How decision is made:	Consider cumulative confidence in the current state estimate	Observe bias between sensor values and estimated values

Table 5.1: Comparison between closed-to-open loop learning and Kalman filtering

the end of the n steps just taken. Since the confidence is accumulated with a product, the desired value will be obtained if each of the past n steps have a confidence equal to the nth root of the desired one.

5.1.5 Relationship to Kalman filters

The closed-to-open-loop architecture has a strong resemblance to the Kalman filters discussed in Section 2.3. A comparison of the two is summarized in Tab. 5.1. Kalman filters do not specifically address the construction of controllers, and F1 corresponds to the controller that Kalman filters assume they get the control signals from. F2 provides the model that is given by $\tilde{F}$ and $\tilde{G}$. It computes the mapping of eq. 2.17. Where Kalman filters have a preset model our architecture uses a changing model.

F3 is the rough equivalent of W in the Kalman filter model. W has a different function in each, though. For Kalman filters it is a continuous value that trades off weighting sensing and estimating to account for the errors in each. In our model the goal is to use the estimation to eliminate sensing when possible, but use it when necessary. Therefore, our W functions as a switch. Our model assumes that estimation errors come from both noise in the plant and inadequacies in F2 while it learns. Sensing is accurate, but undesirable to use because of its cost.

5.2 Experimental simulation results

The goal of the experiments in the last two chapters was to demonstrate the effectiveness of the algorithms on learning to control difficult plants. Here we assume a method of learning to control the plant and seek to demonstrate the algorithm's ability to free up

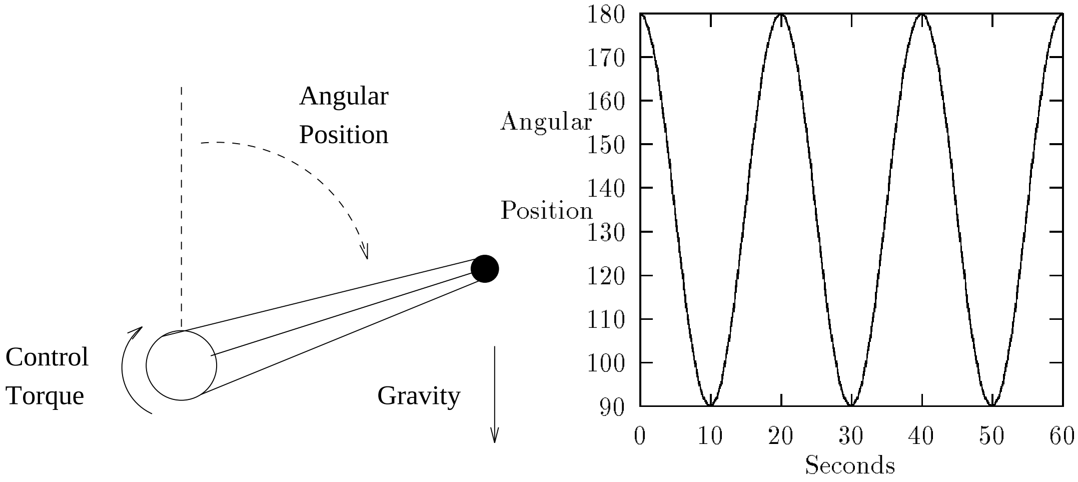

Figure 5.2: Single link tracking task

sensing resources as they become unnecessary. Therefore, we use a much simpler plant in this section. It is a single robot arm in a vertical plane (0^o is straight up) as shown in Fig. 5.2. It is controlled via the application of torque at its base and the output is its angular position and velocity. The state equations are:

x_0 - arm angle
x_1 - arm angular velocity
$\dot{x_0} = x_1$
$\dot{x_1} = c_p(c_1 g \sin(x_0) + f)$

where f is the torque applied at the base and c_p and c_1 are constants (0.1 and 1.0 for these experiments) based on the physical properties of the arm. The constant $c_1 = 1$ is left out of the following equations. The reference curve (shown in Fig. 5.2) for the following experiments is a sine wave moving between 180^o (straight down) and 90^o (horizontal). Each trial consists of three periods of:

$$x_{0,d}(t) = \frac{\pi}{4}\sin(\frac{\pi}{10}t + \frac{\pi}{2}) + \frac{3\pi}{4} \tag{5.4}$$

We generate training data for F1 at each time step with a rule derived from the state equations. First, assume the term $g\sin(x_0)$ to be constant at its value, c_x, for the current x_0. This yields a simple solution to the differential equations under constant torque:

$$x_0(t) = x_0(0) + x_1(0)t + 0.5c_p(c_x + f)t^2 \tag{5.5}$$

$$x_1(t) = x_1(0) + c_p(c_x + f)t \tag{5.6}$$

and we use these to compute the derivatives of position and velocity error with respect to torque:

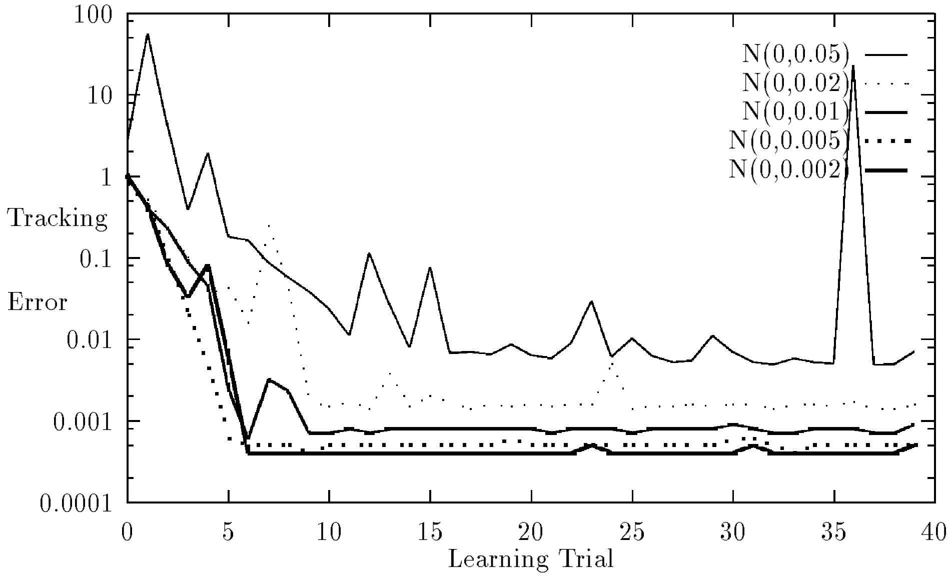

Figure 5.3: Squared tracking error with closed loop only learning at various noise levels

$$\frac{de_0}{df} = -0.5c_p t^2 \tag{5.7}$$

$$\frac{de_1}{df} = -c_p t \tag{5.8}$$

We assume equal penalties for angular position and angular velocity tracking errors and average them in the instantiation of eq. 5.1:

$$f_d = f + 0.5e_0(\frac{2c_p}{t^2}) + 0.5e_1(\frac{c_p}{t}) \tag{5.9}$$

where f_d is the desired torque, f is the torque used, and e_0 and e_1 are the angular position and velocity errors.

First we test the learning of F1 and F2 by setting the open-loop threshold to 1.0 (always sense regardless of how good the learned model is). There is no fixed controller, so the learning starts from a controller that always sends a torque of zero. Finally, zero-mean gaussian noise is added to both state variables at each control cycle. Figs. 5.3 and 5.4 show the tracking error and average prediction confidence resulting from these learning experiments. The data are averages over ten runs and there is a separate curve for each level of noise as indicated in the keys. The tracking error is plotted on a log scale to show the difference in error after learning has converged for each noise level. As expected lower noise levels result in convergence to lower tracking error levels (Fig. 5.3). Similarly, lower noise results in higher confidence in the mapping computed by F2 (Fig. 5.4). The confidence values are slightly deceptive. They only show the average confidence for the states the system actually went through during its runs. As

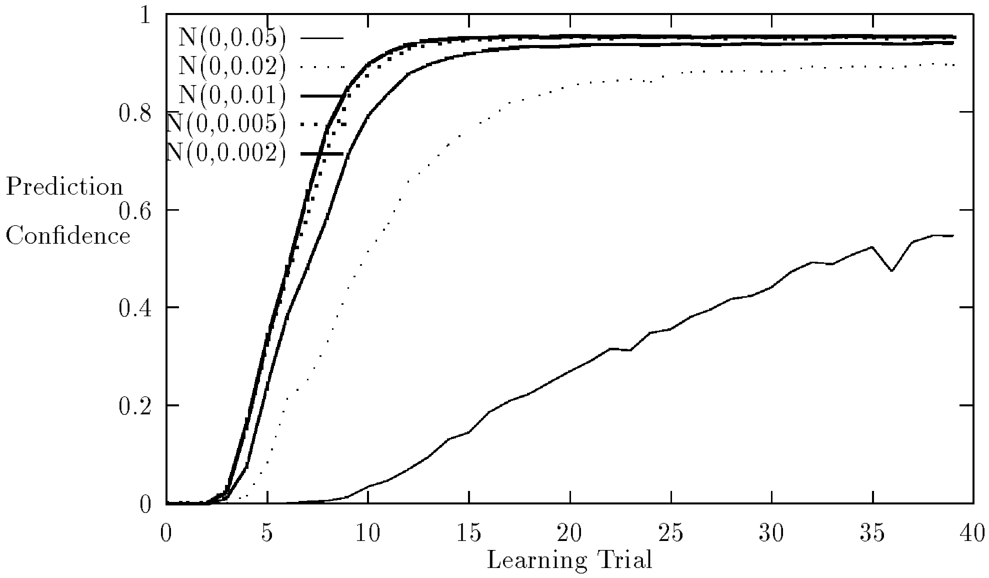

Figure 5.4: Average confidence with closed loop only learning at various noise levels

the system learns it controls the arm better and keeps it in the states it is most confident in, thus further raising the average confidence. These values are shown because they are what will govern the amount of sensing done when the open-loop threshold is lowered.

The fact that the controller has not learned the region of the space where tracking is extremely poor is demonstrated by the late "blip" in the $N(0, 0.05)$ curve. During one of the ten experiments at trial number 36, noise drove the system far off the desired curve and the controller had not yet learned that region of the space. The result was a large error accumulation before the controller brought the system back to the desired trajectory. The lack of controller knowledge in states with a large distance between the current and desired state is far from the desired state could be viewed as a negative for the learner since it has clearly failed to learn the whole space. However, we view it as an indication that it has concentrated the bulk of its learning effort in the region of state space most relevant to highly skilled operation.

Next we show the effects of the open-loop threshold on combined open and closed loop operation. We start with a system that has been trained at the lowest noise level ($N(0, 0.002)$). Then learning is turned off and we operate the system at various threshold values and noise levels. Figs. 5.5 and 5.6 show the tracking error and sensor usage as a function of open-loop threshold and noise level for these experiments. Tracking error increases as the open-loop threshold is lowered and as noise is increased. At the lowest three noise levels error is insignificant for thresholds above 0.6.

Sensor use drops as noise and the open-loop threshold are decreased. The reduction in sensor usage becomes insignificant for thresholds below 0.6. The reason for this is that when the threshold is lowered further the system goes longer between sensing actions and gets farther from the reference curve by the time a sensor measurement is taken. By then the system may be outside the controller's region of expertise and it

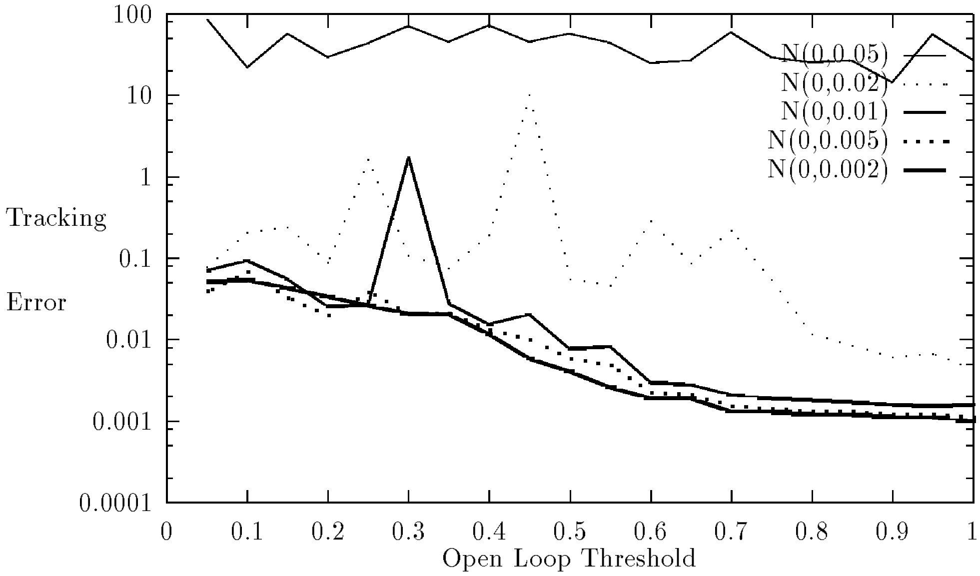

Figure 5.5: Squared tracking error with open-closed loop execution after learning is complete

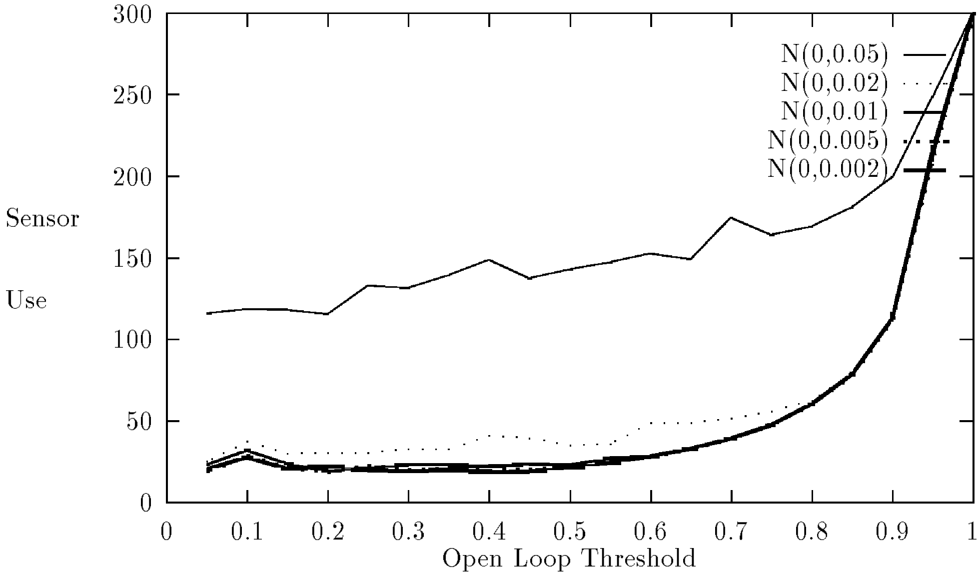

Figure 5.6: Sensing steps used with open-closed loop execution after learning is complete

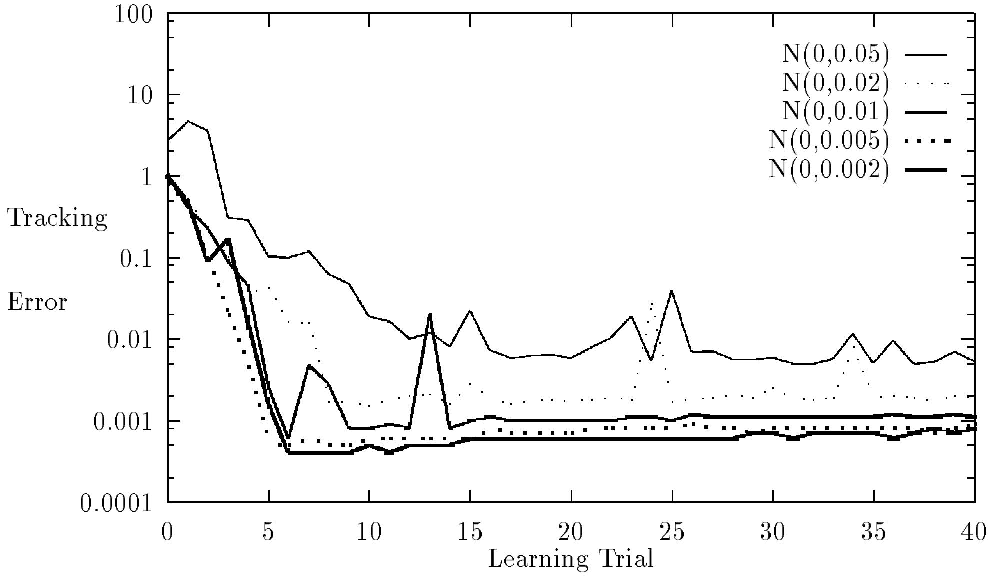

Figure 5.7: Squared tracking error 1-step learning rules

may take several consecutive sensor readings to return it to that region. An analysis of some of the low noise runs shows this phenomenon. At a threshold of 0.5, it almost always requires only one sensor reading to recover each time the confidence falls below the threshold. That yields an average time of 14 steps between sensor readings. At a threshold of 0.1, an average of 1.6 sensor readings are required to recover and there are an average of 21 steps between readings.

Finally, we test the ability of the full system to learn while simultaneously trying to release sensing resources. This is important for systems faced with changing plant characteristics that require them to learn on-line. Since little additional sensor usage reduction is observed for open-loop thresholds below 0.6 and little additional tracking error reduction is observed for thresholds above 0.6 we choose 0.6 as the threshold for these experiments. Figs. 5.7 and 5.8 show the tracking error and sensor usage as learning progresses. Tracking error converges to levels similar to those shown in Fig. 5.3 and in Fig. 5.5 for threshold 0.6. Unfortunately, sensor use converges to a level much higher than that seen in Fig. 5.6. It appears that the problem is the 1-step learning rules. They operate only following a real sensor measurement. As confidence increases to the point where the sensors begin to shut off, learning is also shut off and further progress is inhibited. Despite this problem, usage is still cut by more than half.

The results of experiments using the n-step learning rules are shown in Figs. 5.9 and 5.10. The n-step learning rules are based on the assumption that error and cost is accumulated evenly over the past steps. This assumption holds for short sequences of steps, but can not be expected to hold for longer ones. Note that these figures show twice as many learning trials as the previous ones. Because the learning rules are only loosely correct, it takes longer for the learning to converge. The accuracy reaches levels near that of 1-step learning and the sensor usage as a percentage of the total possble is

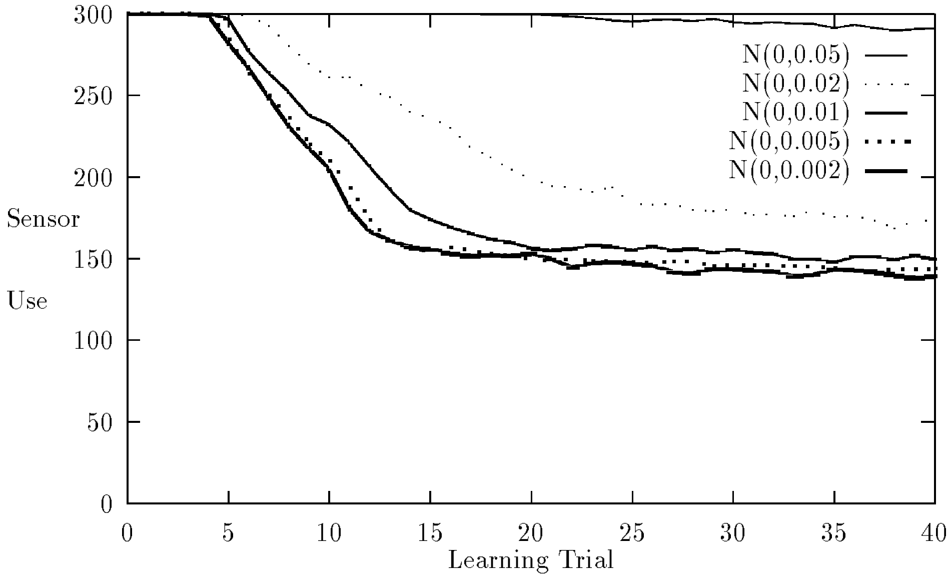

Figure 5.8: Sensing steps used with 1-step learning rules

reduced from 46.3% to 29.3%.

5.3 Summary and future work

In this section we proposed a learning architecture that learns to control and model a plant while giving up sensing resources when it becomes confident in its abilities. One motivation for producing such an architecture is the observation of human athletic performance in team sports. For example, a basketball player has severely constrained visual sensing resources. He would like continuously to monitor the position of the ball, the basket, his teammates, and his opponents while handling the ball or defending an opponent. In order to approach this ability, he learns to execute some skills with almost no visual sensing (dribbling is a good example) even if constant sensing was necessary to learn the skill originally. We believe that learning to perform with minimal sensing will be necessary as complex robots are built to operate in the real world and must exhibit many behaviors simultaneously. Finally, we note that the proposed architecture uses a novel form of intelligent experimentation. Its goal is open-loop execution, but it uses closed loop learning as an efficient search technique to find the right control sequences.

The proposed algorithm does more than reduce sensing. It also *finds* the areas of state space where sensing can be reduced. A complex plant contains both regions of the state space that are highly sensitive to control force and current state and those that are not. The sensitive areas will cause consistently higher prediction errors which will be translated into lower confidence values in those regions. The algorithm uses the approximation in F3 to sort out the regions automatically in order to reduce sensing when possible, but retain it when necessary.

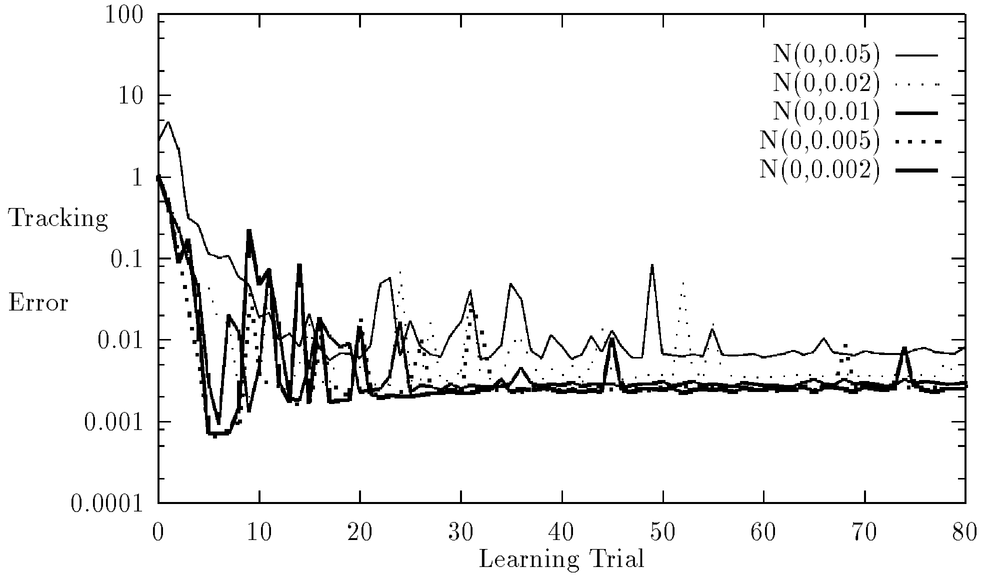

Figure 5.9: Squared tracking error using n-step learning rules

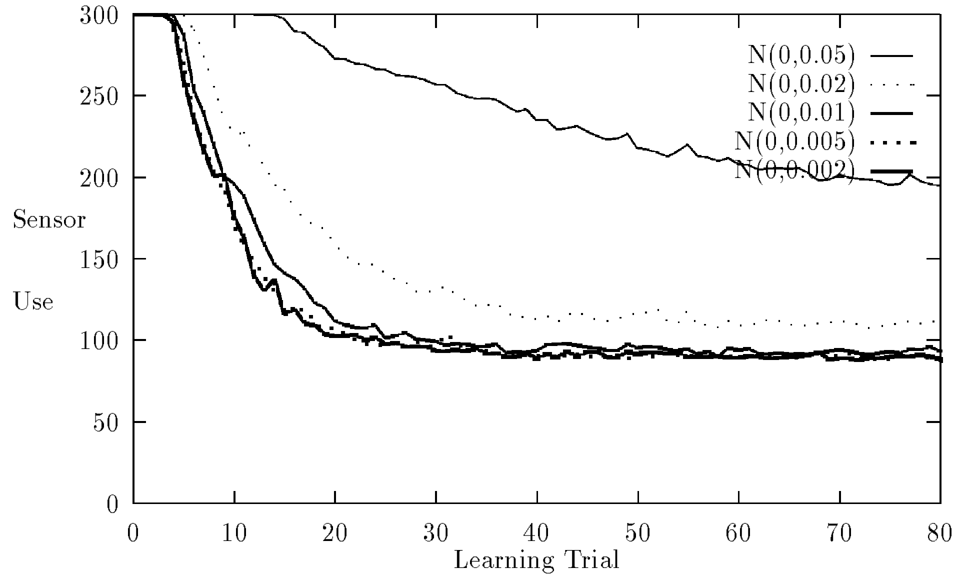

Figure 5.10: Sensing steps used with n-step learning rules

The closed-to-open-loop learning algorithm opens new research possibilities. The most obvious is to improve the current algorithm's reduction of sensor usage when simultaneously learning by improving the n-step learning rules. A sophisticated solution might include techniques such as those from Chapter 3 to find the best sequences of control signals in the presence of sparse sensing.

There are additional issues to be addressed on the incorporation of this algorithm into a complex learning system. We have assumed an implicit, fixed cost of sensing because we treat the ability to turn off the sensors equally at each time step. In a real integrated system, the cost is actually determined by the current demand for sensing resources in each control subsystem. An algorithm must be designed to trade off the goals of continued learning and performance for multiple skills. For instance an easy solution to the problem of premature convergence during learning is to add extra sensor measurements even though F3 does not call for them.

6 Conclusions

The major factor limiting the ability of robots to learn motor skills is acquisition of training data. The data is expensive to obtain because it involves execution of real, physical robots. The most promising way to alleviate this problem is with algorithms that choose their attempted actions intelligently. In this dissertation we proposed several different ways to perform experimentation intelligently and demonstrated the benefits of doing so on sample applications.

6.1 Summary of contributions

6.1.1 Open-loop learning

We presented the Guided Fill In algorithm which efficiently learns open-loop skills. It treats the open-loop skill learning problem as search in a high-dimensional space where only a tiny fraction of the possible actions can be attempted on the real robot. The key to its success is the way it uses information learned in past trials to select which actions it will attempt in future trials. Candidate actions are created from the most successful actions that have already been tried. Intuitively, it is similar to the search technique used by genetic algorithms [Goldberg, 1989]. The difference between GFI and genetic algorithms is that GFI candidate evaluations are considered much more expensive because they require execution on a real robot. Therefore, each candidate is given a measure of success based on the information already learned and only the most promising is evaluated with a real robot trial.

GFI was successful in the robot throwing domain. It was tested on a throwing robot with a flexible link and led to the discovery of a "whipping" motion to throw the ball farther than was possible with a simple forward motion. More importantly, the throwing experiments demonstrated that novel behaviors can be found when a learning system is given many degrees of freedom to experiment with. In order to exploit the extra freedom, efficient search is necessary to find interesting actions.

We presented an algorithm for combining an approximate model with GFI to further improve search efficiency. The algorithm switches between simulated learning using the model (a cheap way to find promising actions), and real robot executions (learns while also producing data to improve the simulation model). We believe combined modeling and learning is the best way to attack many learning control problems.

6.1.2 Closed loop learning

We observed that the performance of local learners is limited by the fact that they do not consider their entire task, but instead modify their controller based only on improving cost for the current step. We proposed "coaching" as a general way to incorporate heuristic information into a closed loop learning controller and discuss its relationship to optimal control. Our experiments showed how a simple smoothing coach can be used to stabilize an inverted pendulum and increase the speed at which it tracks a reference curve.

We also observed that both local learners and single trajectory learners depend on gradient descent not getting stuck in local minima. For simple tasks this is not a problem, but we showed that even a toy inverted pendulum problem can have multiple local minima that can trap the learning controller at undesirable levels of performance. We proposed cooperating coaches as a method of preventing learning from getting stuck. Each coach suggests gradient descent in its own version of the space mapping controller parameters onto performance. The cost functions employed by the coaches may be closely related to the real cost function or only penalize a specific bad characteristic as the smoothing coach does (anything is ok as long as the control is not too jittery). Since each coach is attempting to minimize a different cost function, each has different local minima to contend with. However, as it switches between different coaches, the learning controller avoids getting permanently stuck in any one.

We demonstrated cooperation on the inverted pendulum tracking problem. There is one coach that it is good at reducing the amount of control effort required to track a curve, and a second that is good at learning to track the curve at faster speeds. Through cooperation these two were able to teach a learning controller to combine the best of both. It learns to track the reference curve at high speed and low control effort.

6.1.3 Sensor use

It is desirable to perform motor skills without the use of sensor information (open-loop). Unfortunately, the combinatorics of learning open-loop skills is prohibitive for many tasks. We showed how closed loop learning can be used as a search technique for learning open-loop skills. Closed loop learning can construct a controller and a forward model of the plant. As the controller and the forward model improve, sensors become increasingly unnecessary. Our algorithm automatically gives up its sensor resources as it ceases to depend on them. The result is that the controller learns to perform long sequences of open-loop actions without the costly search that would be necessary to find these sequences in a purely open-loop system.

Our algorithm for reducing sensor use solves another problem in open-loop control. For arbitrary control problems, it may not be easy to determine whether a certain task or reference signal can be handled open-loop. Searching for long open-loop sequences in a purely open-loop system is even more inefficient if it is not known whether any sequence exists that consistently produces the desired result without sensing. The algorithm we presented automatically operates open-loop where it can, and continues to sense in situations where open-loop operation is not possible. This is desirable for plants containing some regions of state space that are very deterministic and other regions for which noise alters the state significantly.

6.2 Future work

We began with the premise that computation is cheap in comparison to real robot trials. This is particularly true for open-loop learning since the system can usually take as long as it wants to determine the next action sequence. GFI, however, does not make good use of that assumption. For the throwing robot it actually generates and evaluates new candidates in far less time than it takes a person to retrieve and reload the ball into the robot's cup. With all the additional time for computation available, it seems likely that a much more sophisticated algorithm could be run between each throw. The modeling work we have done is one possibility, but it may be possible to do more even when no model is available. For example, a more sophisticated algorithm could do a broader statistical evaluation of the entire action space (rather than just a few candidates) in order to find the best possible next throwing action.

Fig. 1.1 shows an entire path leading from open-loop back to open-loop again. The box marked "Closed Loop" represents most of the classic learning and adaptive control work. The box marked "Global Coach" is the topic of Chapter 4 and represents learning based on optimal control. The work in Chapter 5 assumes everything up to the box marked "Global Learner" and discusses a method for following the arc back to open-loop control. The creation of a good global learner is still an open problem. The key to the performance improvements due to global coaching is the use of information about the future reference curve to provide better training information for a learning controller. For the inverted pendulum a global coach applied over a set of reference curves will always suggest that the controller keep the pole more upright. This information will be encoded into the learning controller since it is almost always a good idea and does not depend much on the future of the reference curve.

Consider an example where more detailed information is presented by the coach. Suppose the current state has the pole leaning to the left and the future reference curve also is more toward the left. The coach will suggest leaving the pole leaned left because it will be necessary to move that way anyway. Now suppose the system is in the same state with the same current reference curve, but the future of the reference curve is to the right. The coach, which has the necessary information available, will suggest that the controller stand the pole more upright in preparation for the upcoming move to the right. If the learning controller is using a function approximator with inputs similar to the ones used in Chapters 4 and 5, there is a problem. These inputs are typically the current desired and actual plant state. That creates a mismatch. The global coach is an improvement over local learners because it considers the future reference curve. However, much of the improvement can be lost if the learner does not use a function approximator that also takes information about the future reference curve as its input. If it does not have this information both of the scenarios listed above look the same. Then the function approximator will just average the coach's two suggestions to leave it leaning left, and to stand it more upright for that state.

A possible solution is to add more dimensions to the input space of the function approximator. These dimensions would be used to represent some information about the future reference curve. They would allow the FA to distinguish between the two given scenarios and to learn a different action for each. Combining the global coach with the global learner would yield a system that has the performance of optimal control, while retaining the ability to operate on any reference curve (i.e. it would fit into the lower right quadrant of Fig. 1.2). There are serious obstacles to overcome, though. An increase in dimension can cause an exponential increase in learning time. Therefore, it is necessary to restrict the number of dimensions used to represent the future reference curve. The problem, then, is to determine what variables should be included in the input space of the function approximators. Obvious possibilities are desired positions at specific times in the future or higher derivatives at the present or future.

Finally, mathematical analysis of learning systems is difficult. Most proofs about the performance of learning systems fall into one or both of two categories. The first category of proof makes use of strict assumptions about the plant being controlled. If the plant is simple enough, learning will provide the optimal solution. The second category is proofs of convergence to local minima. These show that application of the learning rule will result in the system reaching a local minima in the cost function. Which minima they fall into depends on where they are started initially. As we have seen in Chapter 4, convergence to local minima is not useful unless something can be said about the relative goodness of the local minima that is reached. There has been little mathematical analysis of the performance of the algorithms in this dissertation and we consider it to be valuable, but difficult future work.

Bibliography

[Aboaf *et al.*, 1988] E. Aboaf, C. Atkeson, and D. Reinkensmeyer, "Task-Level Robot Learning," In *Proceedings of the 88 IEEE Int. Conference on Robotics and Automation*, 1988.

[Aboaf *et al.*, 1989] E. Aboaf, S. Drucker, and C. Atkeson, "Juggling a Tennis Ball More Accurately," In *Proceedings of the 89 IEEE Int. Conference on Robotics and Automation*, 1989.

[Acheson and Mullin, 1993] D. Acheson and T. Mullin, "Upside-down Pendulums," *Nature*, November 1993.

[Albus, 1981] J. Albus, *Brains, Behavior and Robotics*, BYTE Press, 1981.

[Arai *et al.*, 1993] F. Arai, L. Rong, and T. Fukuda, "Trajectory Control of a Flexible Plate Using Neural Network," In *Proceedings of the 93 IEEE Int. Conference on Robotics and Automation*, 1993.

[Arimoto *et al.*, 1984] S. Arimoto, S. Kawamura, and F. Miyazaki, "Bettering Operation of Robots by Learning," *Journal of Robotic Systems*, 1(2), 1984.

[Atkeson, 1991] C. Atkeson, "Using Locally Weighted Regression for Robot Learning," In *Proceedings of the 91 IEEE Int. Conference on Robotics and Automation*, April 1991.

[Atkeson, 1993] C. Atkeson, "Using Local Trajectory Optimizers to Speed Up Global Optimization in Dynamic Programming," In *Advances in Neural Information Processing Systems (NIPS-6)*, 1993.

[Bar-Shalom and Fortmann, 1988] Y. Bar-Shalom and T. Fortmann, *Tracking and Data Association*, Academic Press, 1988.

[Barto *et al.*, 1983] A. Barto, R. Sutton, and C. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[Bellman, 1957] R. Bellman, *Dynamic Programming*, Princeton University Press, 1957.

[Bellman and Dreyfus, 1962] R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, 1962.

[Bianchini and Brown, 1992] R. Bianchini and C. M. Brown, "Parallel Genetic Algorithms on Distributed-Memory Architectures," Technical Report 436, University of Rochester, August 1992.

[Bondi *et al.*, 1988] P. Bondi, G. Casalino, and L. Dambardella, "On the Iterative Learning Control Theory for Robotic Manipulators," *IEEE Journal of Robotics and Automation*, February 1988.

[Branicky, 1991] M. Branicky, "Task-Level Learning: Experiments and Extensions," In *Proceedings of the 91 IEEE Int. Conference on Robotics and Automation*, April 1991.

[Cohn *et al.*, 1994a] D. Cohn, L. Atlas, and R. Ladner, "Improving Generalization with Active Learning," *Machine Learing*, 15(2):201–221, 1994.

[Cohn *et al.*, 1994b] D. Cohn, Z. Ghahramani, and M. Jordan, "Active Learning with Statistical Models," In *Advances in Neural Information Processing Systems 7*, 1994.

[Craig, 1984] J. Craig, "Adaptive Control of Manipulators through Repeated Trials," In *Proceedings of the American Control Conference*, pages 1566–1573, 1984.

[Craig, 1986] J. Craig, *Introduction to Robotics: Mechanics and Control*, Addison-Wesley, 1986.

[Dorf, 1986] R. Dorf, *Modern Control Systems*, Addison-Wesley Publishing Co., 4th edition, 1986.

[Gans, 1992] R. Gans, "On the dynamics of a conservative elastica pendulum," *ASME Journal of Applied Mechanics*, pages 425–430, June 1992.

[Gelb, 1974] A. Gelb, *Applied Optimal Estimation*, The MIT Press, 1974.

[Gelfand *et al.*, 1992] J. Gelfand, M. Flax, R. Endres, S. Lane, and D. Handelman, "Acquisition of Automatic Activity Through Practice: Changes in Sensory Input," In *AAAI-92*, 1992.

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

[Grossberg, 1976] S. Grossberg, "Adaptive Pattern Classification and Universal Recoding: I. Parallel Development and Coding of Neural Feature Detectors," *Biological Cybernetics*, 23:121–134, 1976.

[Hager, 1992] G. Hager, "Constraint Solving Methods and Sensor-based Decision-making," In *Proceedings of the 92 IEEE Int. Conference on Robotics and Automation*, 1992.

[Hager and Mintz, 1991] G. Hager and M. Mintz, "Computational Methods for Task-directed Sensor Data Fusion and Sensor Planning," *International Journal of Robotics Research*, 1991.

[Handelman *et al.*, 1991] D. Handelman, S. Lane, and J. Gelfand, "Goal-Directed Encoding of Task Knowledge for Robotic Skill Acquisition," In *IEEE International Symposium on Intelligent Control*, 1991.

[Hertz *et al.*, 1991] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

[Jacobs *et al.*, 1991] R. Jacobs, M. Jordan, S. Nowlan, and G. Hinton, "Adaptive Mixtures of Local Experts," *Neural Compuation*, 1991.

[Jeannerod, 1988] M. Jeannerod, *The Neural and Behavioural Organization of Goal-Directed Movements*, Oxford University Press, 1988.

[Jordan and Jacobs, 1990] M. Jordan and R. Jacobs, "Learning to Control an Unstable System with Forward Modeling," In *Advances in Neural Information Processing Systems*, 1990.

[Jordan and Rumelhart, 1992] M. Jordan and D. Rumelhart, "Forward Models: Supervised Learning with a Distal Teacher," *Cognitive Science*, 16, 1992.

[Kawamura *et al.*, 1988] S. Kawamura, F. Miyazaki, and S. Arimoto, "Realization of Robot Motion Based on a Learning Method," *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1), 1988.

[Kawato *et al.*, 1987] M. Kawato, K. Furukawa, and R. Suzuki, "A Hierarchical Neural-Network Model for Control and Learning of Voluntary Movement," *Biological Cybernetics*, 57, 1987.

[Kawato *et al.*, 1988] M. Kawato, Y. Uno, M. Isobe, and R. Suzuki, "A Hierarchical Neural Network Model for Voluntary Movement with Application to Robotics," *IEEE Control Systems Magazine*, 8, 1988.

[Kirk, 1970] D. Kirk, *Optimal Control Theory - An Introduction*, Prentice-Hall, 1970.

[Kohonen, 1988] T. Kohonen, *Self-organization and Associative Memory*, Springer-Verlag, 2nd edition, 1988.

[Koza and Keane, 1990] J. Koza and M. Keane, "Cart Centering and Broom Balancing by Genetically Breeding Populations of Control Strategy Programs," In *International Joint Conference on Neural Networks*, 1990.

[Kuperstein, 1988] M. Kuperstein, "Implementation of an Adaptive Visually-guided Neural Controller for Single Postures," In *Proceedings of the American Control Conference*, volume 3, pages 2282–2287, 1988.

[Kuperstein and Rubenstein, 1989] M. Kuperstein and J. Rubenstein, "Implementation of an Adaptive Neural Controller for Sensory-motor Coordination," *IEEE Control Systems Magazine*, 9(3), 1989.

[Lane *et al.*, 1992] S. Lane, D. Handelman, and J. Gelfand, "Theory and Development of Higher-Order CMAC Neural Networks," *IEEE Control Systems Magazine*, April 1992.

[Langley *et al.*, 1994] P. Langley, W. Iba, and J. Shrager, "Reactive and Automatic Behavior in Plan Execution," In *Proceedings of the Second International Conference on AI Planning Systems*, 1994.

[Leitmann, 1981] G. Leitmann, *The Calculus of Variations and Optimal Control*, Plenum Press, 1981.

[Lewis, 1986] F. Lewis, *Optimal Control*, John Wiley and Sons, 1986.

[McCallum, 1994] A. McCallum, "Instance-based State Identification for Reinforcement Learning," In *Advances in Neural Information Processings Systems 7 (NIPS-94)*, 1994.

[McGeer, 1990] T. McGeer, "Passive Dynamic Walking," *International Journal of Robotics Research*, 1990.

[Mel, 1990] B. Mel, *Connectionist Robot Motion Planning: A Neurally Inspired Approach to Visually Guided Reaching*, Academic Press, 1990.

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

[Miyamoto *et al.*, 1988] H. Miyamoto, M. Kawato, T. Setoyama, and R. Suzuki, "Feedback-Error-Learning Neural Network for Trajectory Control of a Robotic Manipulator," *Neural Networks*, 1, 1988.

[Moody and Darken, 1989] J. Moody and C. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, pages 281–294, 1989.

[Moore, 1990] A. Moore, *Efficient Memory-Based Learning for Robot Control*, PhD thesis, University of Cambridge, November 1990.

[Moore, 1991] A. Moore, "Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-Valued State-Spaces," In *Proceedings of the 8th International Workshop on Machine Learning*, 1991.

[Moore, 1993] A. Moore, "The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State Spaces," In *Advances in Neural Information Processings Systems 6 (NIPS-93)*, 1993.

[Moore and Atkeson, 1993] A. Moore and C. Atkeson, "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time," *Machine Learning*, 13(1):103–130, 1993.

[Newton and Xu, 1993] R. Newton and Y. Xu, "Real-Time Implementation of Neural Network Learning Control of a Flexible Space Manipulator," In *Proceedings of the 93 IEEE Int. Conference on Robotics and Automation*, 1993.

[Ogata, 1987] K. Ogata, *Discrete-Time Control Systems*, Prentice-Hall, 1987.

[Pearlmutter, 1989] B. Pearlmutter, "Learning State Space Trajectories in Recurrent Neural Networks," *Neural Computation*, pages 263–269, 1989.

[Pineda, 1989] F. Pineda, "Recurrent Backpropagation and the Dyanamical Approach to Adaptive Neural Computation," *Neural Computation*, pages 161–172, 1989.

[Poggio and Girosi, 1989] T. Poggio and F. Girosi, "A Theory of Networks for Approximation and Learning," Technical Report 1140, MIT AI Lab, 1989.

[Poggio and Girosi, 1990] T. Poggio and F. Girosi, "Regularization Algorithms for Learning that are Equivalent to Multilayer Networks," *Science*, pages 978–982, February 1990.

[Pomerleau, 1989] D. Pomerleau, "ALVINN: An Autonomous Land Vehicle in a Neural Network," In *Advances in Neural Information Processing Systems 1*, 1989.

[Press *et al.*, 1992] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1992.

[Raibert, 1977] M. Raibert, "Analytical equations vs table look-up for manipulation: a unifying concept," In *Proceedings of the IEEE Conference on Decision and Control*, 1977.

[Raibert, 1978] M. Raibert, "A model for sensorimotor control and learning," *Biological Cybernetics*, 1978.

[Rimey and Brown, 1994] R. Rimey and C. Brown, "Control of Selective Perception Using Bayes Nets and Decision Theory," *International Journal of Computer Vision*, 1994.

[Ritter *et al.*, 1992] H. Ritter, T. Martinetz, and K. Schulten, *Neural Computation and Self-Organizing Maps*, Addison-Wesley, 1992.

[Rizzi and Koditschek, 1992] A. Rizzi and D. Koditschek, "Progress in Spatial Robot Juggling," In *Proceedings of the 92 IEEE Int. Conference on Robotics and Automation*, 1992.

[Rizzi and Koditschek, 1993] A. Rizzi and D. Koditschek, "Further Progress in Robot Juggling: The Spatial Two-Juggle," In *Proceedings of the 93 IEEE Int. Conference on Robotics and Automation*, 1993.

[Rumelhart *et al.*, 1986] D. Rumelhart, G. Hinton, and R. Williams, *Learning Internal Representations by Error Propagation*, volume 1, chapter 8, MIT Press, 1986.

[Rumelhart and Zipser, 1986] D. Rumelhart and D. Zipser, *Feature Discovery by Competitive Learning*, volume 1, pages 151–193, MIT Press, 1986.

[Sakaguchi *et al.*, 1993] T. Sakaguchi, M. Fujita, H. Watanabe, and F. Miyazaki, "Motion Planning and Control for a Robot Performer," In *Proceedings of the 93 IEEE Int. Conference on Robotics and Automation*, 1993.

[Salganicoff, 1992] M. Salganicoff, *Learning and Forgetting for Perception-Action: A Projection-Pursuit and Density Adaptive Approach*, PhD thesis, University of Pennsylvania, 1992.

[Salganicoff, 1993] M. Salganicoff, "Density-Adaptive Learning and Forgetting," In *Tenth International Conference on Machine Learning*, 1993.

[Sanger, 1994] T. Sanger, "Neural Network Learning Control of Robot Manipulators using Gradually Increasing Task Difficulty," *IEEE Transactions on Robotics and Automation*, 10:323–333, 1994.

[Schaal and Atkeson, 1993a] S. Schaal and C. Atkeson, "Assessing the Quality of Learned Local Models," In *Advances in Neural Information Processing Systems (NIPS-6)*, 1993.

[Schaal and Atkeson, 1993b] S. Schaal and C. Atkeson, "Open Loop Stable Control Strategies for Robot Juggling," In *Proceedings of the 93 IEEE Int. Conf. on Robotics and Automation*, 1993.

[Schaal and Atkeson, 1994] S. Schaal and C. Atkeson, "Robot Juggling: An Implementation of Memory-Based Learning," *IEEE Control Systems Magazine*, 14(1), 1994.

[Schneider, 1994] J. Schneider, "High Dimension Action Spaces in Robot Skill Learning," In *Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[Schneider and Brown, 1992] J. Schneider and C. Brown, "Robot Skill Learning and the Effects of Basis Function Choice," Technical Report 437, University of Rochester, September 1992.

[Schneider and Brown, 1993] J. Schneider and C. Brown, "Robot Skill Learning, Basis Functions, and Control Regimes," In *Proceedings of the 93 IEEE Int. Conf. on Robotics and Automation*, pages 403–410, 1993.

[Schneider and Brown, 1994] J. Schneider and C. Brown, "Task Level Training Signals for Learning Controllers," In *9th IEEE International Symposium on Intelligent Control*, pages 45–50, 1994.

[Schneider and Gans, 1994] J. Schneider and R. Gans, "Efficient Search for Robot Skill Learning: Simulation and Reality," In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, 1994.

[Simard, 1991] P. Simard, *Learning State Space Dynamics in Recurrent Networks*, PhD thesis, University of Rochester, June 1991.

[Sutton, 1984] R. Sutton, *Temporal Credit Assignment in Reinforcement Learning*, PhD thesis, University of Massachusetts, 1984.

[Sutton, 1990] R. Sutton, "First Results with DYNA, an Intergrated Architecture for Learning, Planning, and Reacting," In *AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[Tan, 1992] M. Tan, "Learning a Cost Sensitive Internal Representation for Reinforcement Learning," In *Proceedings of the Eigth International Workshop on Machine Learning*, pages 358–362, 1992.

[Tan, 1993] M. Tan, "Cost-Sensitive Learning of Classification Knowledge and Its Applications in Robotics," *Machine Learning*, 13(1):7–32, 1993.

[Watkins, 1989] C. Watkins, *Learning from Delayed Rewards*, PhD thesis, Cambridge University, 1989.

[Watkins and Dayan, 1992] C. Watkins and P. Dayan, "Q Learning," *Machine Learning*, pages 279–292, 1992.

[Werbos, 1974] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Master's thesis, Harvard University, 1974.

[White and Jordan, 1992] D. White and M. Jordan, "Optimal Control: A Foundation for Intelligent Control," In *Handbook of Intelligent Control*, chapter 6. Van Nostrand Reinhold, 1992.

[White and Sofge, 1992] D. White and D. Sofge, *Handbook of Intelligent Control*, Van Nostrand Reinhold, 1992.

[Wixson and Ballard, 1994] L. Wixson and D. Ballard, "Using Intermediate Objects to Improve the Efficiency of Visual Search," *International Journal of Computer Vision*, 1994.

A Numerical Estimation of the Optimal Curve for Ball Throwing

The simulated robot for 1-d throwing has a 12-d action space. There are two pieces of information that can be used to reduce it to a 4-d space and thus make the learning faster. The first is to note that the landing point of the ball depends only on the position and velocity of the end effector at the time of release. The second is that for a given final end effector position and velocity, it is possible to use the effort equation (the second term in eq. 3.4) and the kinematic equations to get a closed form solution for the best trajectory given an ending position and velocity. The following constrained optimization is done:

Minimize:

$$F = \sum_{i=1}^{joints} (x_{i,1}^2 + \sum_{j=2}^{timesteps} (x_{i,j} - x_{i,j-1})^2)$$

Subject to:

$$G_1 = startpos_1 - finalpos_1 + \sum_{j=1}^{timesteps} x_{1,j} = 0$$

$$G_2 = startpos_2 - finalpos_2 + \sum_{j=1}^{timesteps} x_{2,j} = 0$$

Note that the final velocity in each of the two time sequences is a given (along with the start and final position). Using Lagrangian multipliers, it is straightforward to solve for the remaining joint velocities. The solution reduces the action space to 4-d (two ending joint positions and two ending joint velocities).

The reduction to a 4-d space makes it possible to do a brute force determination of the optimal solution to the ball throwing problem. The final joint position values were quantized into 0.25 degree increments and the final joint velocity values were quantized into 0.10 degree increments. The output distance was quantized into bins of size 10. All possible trajectories were attempted using this discretization and each was placed in the appropriate distance bin. The lowest effort trajectory for each bin was recorded as the optimal and used to create Fig. 3.15.