Approximate Solutions to Markov Decision Processes

Geoffrey J. Gordon

June 1999

CMU-CS-99-143

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Tom Mitchell, Chair
Andrew Moore
John Lafferty
Satinder Singh Baveja, AT&T Labs Research

This research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Nos. F30602-97-1-0215 and F33615-93-1-1330, by the National Science Foundation (NSF) under Grant No. BES-9402439, and by an NSF Graduate Research Fellowship. The views and conclusions expressed in this publication are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, or the U.S. government.

Keywords: machine learning, reinforcement learning, dynamic programming, Markov decision processes (MDPs), linear programming, convex programming, function approximation, worst-case learning, regret bounds, statistics, fitted value iteration, convergence of numerical methods

Abstract

One of the basic problems of machine learning is deciding how to act in an uncertain world. For example, if I want my robot to bring me a cup of coffee, it must be able to compute the correct sequence of electrical impulses to send to its motors to navigate from the coffee pot to my office. In fact, since the results of its actions are not completely predictable, it is not enough just to compute the correct sequence; instead the robot must sense and correct for deviations from its intended path.

In order for any machine learner to act reasonably in an uncertain environment, it must solve problems like the above one quickly and reliably. Unfortunately, the world is often so complicated that it is difficult or impossible to find the optimal sequence of actions to achieve a given goal. So, in order to scale our learners up to real-world problems, we usually must settle for approximate solutions.

One representation for a learner's environment and goals is a Markov decision process or MDP. MDPs allow us to represent actions that have probabilistic outcomes, and to plan for complicated, temporally-extended goals. An MDP consists of a set of states that the environment can be in, together with rules for how the environment can change state and for what the learner is supposed to do.

One way to approach a large MDP is to try to compute an approximation to its optimal state evaluation function, the function which tells us how much reward the learner can be expected to achieve if the world is in a particular state. If the approximation is good enough, we can use a shallow search to find a good action from most states. Researchers have tried many different ways to approximate evaluation functions. This thesis aims for a middle ground, between algorithms that don't scale well because they use an impoverished representation for the evaluation function and algorithms that we can't analyze because they use too complicated a representation.

Acknowledgements

This work would not have been possible without the support of my thesis committee and the rest of the faculty at Carnegie Mellon. Thanks in particular to my advisor Tom Mitchell and to Andrew Moore for helping me to see both the forest and the trees, and to Tom Mitchell for finding the funding to let me work on the interesting problems rather than the lucrative ones. Thanks also to John Lafferty, Avrim Blum, Steven Rudich, and Satinder Singh Baveja, both for the advice they gave me and the knowledge they taught me.

Finally, and most of all, thanks to my wife Paula for supporting me, encouraging me, listening to my complaints, making me laugh, laughing at my jokes, taking the time just to sit and talk with me, and putting up with the time I couldn't spend with her.

Contents

Chapter 1

INTRODUCTION

One of the basic problems of machine learning is deciding how to act in an uncertain world. For example, if I want my robot to bring me a cup of coffee, it must be able to compute the correct sequence of electrical impulses to send to its motors to navigate from the coffee pot to my office. In fact, since the results of its actions are not completely predictable, it is not enough just to compute the correct sequence; instead the robot must sense and correct for deviations from its intended path.

In order for any machine learner to act reasonably in an uncertain environment, it must solve problems like the above one quickly and reliably. Unfortunately, the world is often so complicated that it is difficult or impossible to find the optimal sequence of actions to achieve a given goal. So, in order to scale our learners up to real-world problems, we usually must settle for approximate solutions.

One representation for a learner's environment and goals is a Markov decision process or MDP. MDPs allow us to represent actions that have probabilistic outcomes, and to plan for complicated, temporally-extended goals. An MDP consists of a set of states that the environment can be in, together with rules for how the environment can change state and for what the learner is supposed to do.

Given an MDP, our learner can in principle search through all possible sequences of actions up to some maximum length to find the best one. In practice the search will go faster if we know a good heuristic evaluation function, that is, a function which tells us approximately how good or bad it is to be in a given state. For small MDPs we can compute the best possible heuristic evaluation function. With this optimal evaluation function, also called the value function, a search to depth one is sufficient to compute the optimal action from any state.

One way to approach a large MDP is to try to compute an approximation to its value function. If the approximation is good enough, a shallow search will be able to find a good action from most states. Researchers have tried many different ways to compute value functions, ranging from simple approaches based on dividing the states into bins and assigning the same value to all states in each bin, to complicated approaches involving neural networks and stochastic approximation. Unfortunately, in general the simple approaches don't scale well, while the complicated approaches are difficult to analyze and are not guaranteed to reach a reasonable solution.

This thesis aims for a middle ground, between algorithms that don't scale well because they use an impoverished representation for the value function and algorithms that we can't analyze because they use too complicated a representation. All of the research in this thesis was motivated by the attempt to find algorithms that can use a reasonably rich representation for value functions but are still guaranteed to converge. In particular, we looked for algorithms that can represent the value function as a linear combination of arbitrary but fixed basis functions. While the algorithms we describe do not quite achieve this goal, they do represent a significant advance over the previous state of the art.

There are three main parts to this thesis. In Chapter 2 we will describe an approach that lets us approximate an MDP's value function using linear interpolation, nearest-neighbor, or other similar methods. In Chapter 3 we will step back and consider a more general problem, the problem of learning from a sequence of training examples when we can't make distributional assumptions. This chapter will also serve as an introduction to the theory of convex optimization. Finally, in Chapter 4, we will apply the theory of linear programming and convex optimization to the problem of approximating an MDP's value function. Chapters 2 and 4 each contain experimental results from different algorithms for approximating value functions. In addition to the three groups of results listed above, this thesis also contains references to related work (in Chapter 5) and a concluding summary (in Chapter 6).

These three threads of research work together towards the goal of finding approximate value functions for Markov decision processes. The contribution of Chapter 2 is the most direct: it enlarges the class of representations we can use for approximate value functions to include methods such as k-nearest-neighbor, multilinear interpolation, and kernel regression, for which there were previously no known convergent algorithms. While Chapter 3 does not mention MDPs directly, it treats the problem of learning without a fixed sampling distribution or independent samples, which is one of the underlying difficulties in learning about MDPs. Finally, Chapter 4 presents a framework for designing value function approximating algorithms that allow even more general representations than those of Chapter 2.

In more detail, Chapter 2 describes a class of function approximation architectures (which contains, *e.g.*, k-nearest-neighbor and multilinear interpolation) for which an algorithm called fitted value iteration is guaranteed to converge. The contributions of Chapter 2 include discovering this class and deriving convergence rates and error bounds for the resulting algorithms. The contributions also include an improved theoretical understanding of fitted value iteration via a reduction to exact value iteration, and experimental results showing that fitted value iteration is capable of complex pattern recognition in the course of solving an MDP.

Chapter 3 presents results about the data efficiency of a class of learning algorithms (which contains, *e.g.*, linear and logistic regression and the weighted majority algorithm) when traditional statistical assumptions do not hold. The type of performance result we prove in Chapter 3 is called a worst-case regret bound, because it holds for all sequences of training examples and because it bounds the regret of the algorithm or the difference between its performance and a defined standard of comparison. Since one of the difficulties with learning about Markov decision processes is that the training samples are often not independent or identically distributed, better worst-case bounds on learning algorithms are a first step towards using these algorithms to learn about MDPs. The contributions of Chapter 3 are providing a unified framework for deriving worst-case regret bounds and applying this framework to prove regret bounds for several well-known algorithms. Some of these regret bounds were known previously, while others are new.

Chapter 4 explores connections between the problem of solving an MDP and the problems of convex optimization and statistical estimation. It then proposes algorithms motivated by these connections, and describes experiments with one of these algorithms. While this new algorithm does not improve on the best existing algorithms, the motivation behind it may help with the design of other algorithms. The contributions of this chapter include bringing together results about MDPs, convex optimization, and statistical estimation; analyzing the shortcomings of existing value-function approximation algorithms such as fitted value iteration and linear programming; and designing and experimenting with new algorithms for solving MDPs.

In the remainder of this introduction we will define Markov decision processes and describe an algorithm for finding exact solutions to small MDPs. This algorithm, called value iteration, will be our starting point for deriving the results of Chapter 2; and the underlying motivation for value iteration, namely representing the value function as the solution of a set of nonlinear equations called the Bellman equations, will provide the starting point for the results of Chapter 4.

1.1 Markov decision processes

A Markov decision process is a representation of a planning problem. Figure 1.1 shows a simple example of an MDP. This MDP has four states: the agent starts at the leftmost state, then has the choice of proceeding to either of the two middle states. If it chooses the upper state it is charged a cost of 1 unit; if it chooses the lower, it is charged a cost of 2 units. In either case the agent must then visit the final state at a cost of 1 unit, after which the problem ends.

The MDP of Figure 1.1 is small and deterministic. Other MDPs may be much larger and may have actions with stochastic outcomes. For example, later on we will consider an MDP which has more than 10^{50} states. We are also interested in MDPs with infinitely many states, although we will usually replace such an MDP by a finite approximation.

More formally, a Markov decision process is a tuple $(S, A, \delta, c, \gamma, S_0)$. The set S is the state space; the set A is the action space. At any time t, the environment is in some state $x_t \in S$. The agent perceives x_t, and is allowed to choose an action $a_t \in A$. (If $|A| = 1$, so that the agent has only one choice on each step, the model is called a Markov process instead of a Markov decision process.) More generally, the available actions may depend on x_t; if this is the case the agent's choice is restricted to some set $A(x_t) \subseteq A$. The transition function δ (which may be probabilistic) then acts on x_t and a_t to produce a next state x_{t+1}, and the process repeats. The state x_{t+1} may be either an element of S or the symbol $\odot$ which signifies that the problem is over; by definition $\delta(\odot, a) = \odot$ for any $a \in A(\odot)$. A sequence of states and actions generated this way is called a trajectory. S_0 is a distribution on S which gives the probability of being in each state at time 0. The cost function, c (which may be probabilistic, but must have finite mean and variance), measures how well the agent is doing: at each time step t, the agent incurs a cost $c(x_t, a_t)$. By definition $c(\odot, a) = 0$ for any a. The agent must act to minimize the expected discounted cost $\mathbb{E}(\sum_{t=0}^{\infty} \gamma^t c(x_t, a_t))$;

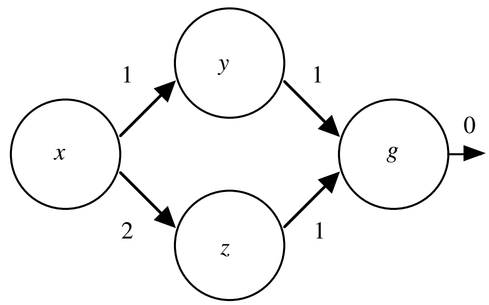


Figure 1.1: A simple MDP.

$\gamma \in [0, 1]$ is called the discount factor.

A function π which assigns an action to every state is called a policy. Following the policy π means performing action $\pi(i)$ when in state i. If we write p_{aij} for the probability of reaching state j when performing action a from state i, then we can define a matrix P_π whose i, jth element is $p_{\pi(i),i,j}$. P_π is called the transition probability matrix for π.

A deterministic undiscounted MDP can be regarded as a weighted directed graph, just like the example in Figure 1.1: each state of the MDP corresponds to a node in the graph, while each state-action pair corresponds to a directed edge. There is an edge from node x to node y iff there is some action a so that $\delta(x, a) = y$; the weight of this edge is $c(x, a)$. In Figure 1.1 we have adopted the convention that an edge coming out of some node that points to nowhere corresponds to a transition from that node to $\odot$.

We can represent a graph like the one in Figure 1.1 with an adjacency matrix and a cost vector. The adjacency matrix E has one row for each edge and one column for each node. The row for an edge (i, j) has a -1 in column i and a $+1$ in column j, and all other elements 0. The cost vector c has one element for each edge; the element for an edge $(i, \delta(i, a))$ is equal to $c(i, a)$.

The adjacency matrix E is related to the transition probability matrices P_π: for any deterministic policy π, $P_\pi - I$ is a submatrix of E. Similarly, c_π (defined to be the vector whose ith element is $c(i, \pi(i))$) is a subvector of c. In fact, if we think of a policy as a subset of the edges containing exactly one edge leading out of each state, then $P_\pi - I$ is the submatrix of E that results from deleting all rows that correspond to edges not in π.

We can generalize the idea of an adjacency matrix to stochastic or discounted MDPs: the idea is that $\gamma P_\pi - I$ should still always be a submatrix of E. So, we define E to be a matrix with one row for every state-action pair in the MDP. If action a executed in state i has probability p_{aij} of moving the agent to state j, then we define the jth entry in the row of E for state i and action a to be either γp_{aij} (if $i \neq j$) or $\gamma p_{aij} - 1$ (if $i = j$). Similarly, we can generalize the cost vector by setting c to be the vector whose element in the row corresponding to state i and action a is $\mathbb{E}(c(i, a))$.

Often we will write the adjacency matrix E without the column corresponding to $\odot$ and without the rows corresponding to transitions out of $\odot$. This causes no loss of information, since any missing probability mass in a transition may be assumed to belong to $\odot$, and since the transitions out of $\odot$ are determined by the definition of an MDP.

1.2 MDP examples

In addition to the simple example from Figure 1.1, Markov decision processes can represent much larger, more complicated planning problems. Some of the MDPs that researchers have tried to solve are:

- Factory production planning. In this problem different states correspond to different inventory levels of various products or different arrangements of the production lines, while actions correspond to possible rearrangements of the production lines. The cost function includes money spent on raw materials, rent paid for warehouse space, and profits earned from selling products.

- Control of a robot arm. In this problem the state encodes the position, joint angles, and joint velocities of the arm, as well as the locations of obstacles in the workspace. Actions specify joint torques, and the cost function includes bonuses for bringing the arm close to its target configuration and penalties for collisions or jerky motion.

- Elevator scheduling. The state for this problem includes such information as the locations of the elevators and whether each call button has been pressed. The actions are to move the elevators from floor to floor and open and close their doors, and the cost function penalizes the learner for making people wait too long before being picked up or let off.

- The game of Tetris. We discuss this MDP in more detail in Chapter 4. Its state includes the current configuration of empty and filled squares on the board as well as the type of piece to be placed next. The actions specify where to place the current piece, and the reward for each transition is equal to the change in the player's score.

These MDPs are all too large to solve exactly; for example, the version of Tetris we describe in Chapter 4 has more than 10^{50} states, while the robot arm control problem has infinitely many states because it includes real-valued variables such as joint angles.

1.3 Value iteration

If our MDP is sufficiently small, we can find the exact optimal controller by any of several methods, for example value iteration, policy iteration, or linear programming (see [Ber95]). These methods are based on computing the so-called value, evaluation, or cost-to-go function, which is defined by the recursion

$$v(x) = \min_{a \in A} \mathbb{E}(c(x,a) + \gamma v(\delta(x,a)))$$

(If $\gamma = 1$ we will need to specify one or more base cases such as $v(\odot) = 0$ to define a unique value function.) This recursion is called the Bellman equation. If we know the value function, it is easy to compute an optimal action from any state: any a which achieves the minimum in the Bellman equation will do. For example, the value function for the MDP of Figure 1.1 is $(x, y, z, g) = (2, 1, 1, 0)$. The edge from x to y achieves the minimum in the Bellman equation, while the edge from x to z does not; so, the optimal action from state x is to go to y.

Value iteration works by treating the Bellman equation as an assignment. That is, it picks an arbitrary initial guess $v^{(0)}$, and on the ith step it sets

$$v^{(i+1)}(x) = \min_{a \in A} \mathbb{E}(c(x,a) + v^{(i)}(\delta(x,a))) \tag{1.1}$$

for every $x \in X$. For the special case of deterministic undiscounted MDPs, the problem of finding an optimal controller is just the single-destination minimum-cost paths problem, and value iteration is called the Bellman-Ford algorithm.

To save writing one copy of Equation 1.1 for each state, we define the vector operator T so that

$$v^{(i+1)} = T(v^{(i)})$$

In other words, T performs one step of value iteration on its argument, updating the value of every state in parallel according to the Bellman equation. A step of value iteration is called a backup, and T is called the backup operator.

A greedy policy for a given value function is one in which, for all x, $\pi(x)$ achieves the minimum in the right-hand side of the Bellman equation. Given a policy π, define T_π so that

$$[T_\pi(v)]_x = \mathbb{E}(c(x, \pi(x)) + [v]_{\delta(x,\pi(x))})$$

where the notation $[v]_x$ stands for component x of the vector v. T_π is called the backup operator for π. If π is greedy for v, then $Tv = T_\pi v$.

The operator T_π is affine, that is, there is a matrix γP_π and a vector c_π so that $T_\pi v = \gamma P_\pi v + c_\pi$. In fact, P_π is the transition probability matrix for π, and c_π is the cost vector for π. That is, the elements of c_π are the costs $c(x, \pi(x))$ for each state x, while the row of P_π which corresponds to state x contains the probability distribution for the state x_{t+1} given that $x_t = x$ and that we take action $\pi(x)$.

If $\gamma < 1$, the operator T is a contraction in max norm. That is, if u and v are estimates of the value function, then $||Tu - Tv||_\infty \leq \gamma ||u - v||_\infty$. If $\gamma = 1$, then under mild conditions T is a contraction in some weighted max norm. In either case, by the contraction mapping theorem (see [BT89]), value iteration converges to the unique solution of the Bellman equations.

Chapter 2

FITTED VALUE ITERATION

Up to this point, we have described how to find the exact solution to a Markov decision process. Unfortunately, we can only find exact solutions for small MDPs. For larger MDPs, we must resort to approximate solutions.

Any approximate solution must take advantage of some prior knowledge about the MDP: in the worst case, when we don't know anything about which states are similar to which others, we have no hope of even being able to represent a good approximate solution. Luckily, if we have to solve a large MDP in practice, we usually know something about where it came from. For example, an MDP with $10^{10} + 1$ states is probably too large to solve exactly with current computers; but if we know that these states are the dollar amounts in one-cent increments between zero and a hundred million, we can take advantage of the fact that a good action from the state \$1053.76 is probably also a good action from the state \$1053.77. Similarly, we usually can't solve an MDP with infinitely many states exactly, but if we know the states are the positions between 0 and 1m we can take advantage of the fact that a motion of 1nm is unlikely to matter very much.

The simplest and oldest method for finding approximate value functions is to divide the states of the MDP into groups, pick a representative state from each group, and pretend that the states in each group all have the same value as their representative. For example, in the MDP with states between 0 and 1m, one group could be the states from 0 to 1cm with representative 0.5cm, the next could be the states from 1cm to 2cm with representative 1.5cm, and so forth, for a total of 100 groups. If a 1cm resolution turned out to be too coarse in some interval, say between 33cm and 34cm, we could replace that group with a larger number of finer divisions, say 330mm to 331mm, 331mm to 332mm, and so forth, giving a total of 109 groups.

Once we have divided the states into groups we can run value iteration just as before. If we see a transition that ends in a non-representative state, say one that takes us to the state 1.6cm, we look up the value of the appropriate representative, in this case 1.5cm. This way we only have to store and update the values for the representative states, which means that we only have to pay attention to transitions that start in the representative states. So, value iteration will run much faster than if we had to examine all of the values and all of the transitions.

This method for finding approximate value functions is called state aggregation. It can work well for moderate-sized MDPs, but it suffers from a problem: if we choose to divide each axis of a d-dimensional continuous state space into k partitions, we will wind up with k^d states in our discretization. Even if k and d are both relatively small we can wind up with a huge number of states. For example, if we divide each of six continuous variables into a hundred partitions each, the result is 10^{12} distinct states. This problem is called the curse of dimensionality, since the number of states in the discretization is exponential in d.

To avoid the curse of dimensionality, we would like to have an algorithm that works with more flexible representations than just state aggregation. For example, rather than setting a state's value to that of a single representative, we might prefer to interpolate linearly between a pair of neighboring representatives; or, in higher dimensions, we might want to set a state's value to the average of the k nearest representatives. This kind of flexibility can let us get away with fewer representatives and so solve larger problems.

One algorithm that can take advantage of such representations is fitted value iteration, which is the subject of this chapter. Fitted value iteration generalizes state aggregation to handle representations like linear interpolation and k-nearest-neighbor.

In fitted value iteration, we interleave steps of value iteration with steps of function approximation. It will turn out that, if the function approximator satisfies certain conditions, we will be able to prove convergence and error bounds for fitted value iteration. If, in addition, the function approximator is linear in its parameters, we will be able to show that fitted value iteration on the original MDP is equivalent to exact value iteration on a smaller MDP embedded within the original one.

The conditions on the function approximator allow such widely-used methods as k-nearest-neighbor, local weighted averaging, and linear and multilinear interpolation; however, they rule out all but special cases of linear regression, local weighted regression, and neural net fitting. In later chapters we will talk about ways to use more general function approximators.

Most of the material in this chapter is drawn from [Gor95a] and [Gor95b]. Some of this material was discovered simultaneously and independently in [TV94]. A related algorithm which learns online (that is, by following trajectories in the MDP and updating states only as they are visited, in contrast to the way fitted value iteration can update states in any order) is described in [SJJ95].

2.1 Discounted processes

In this section, we will consider only discounted Markov decision processes. Section 2.2 generalizes the results to nondiscounted processes.

Suppose that T_M is the parallel value backup operator for a Markov decision process M, as defined in Chapter 1. In the basic value iteration algorithm, we start off by setting v_0 to some initial guess at M's value function. Then we repeatedly set v_{i+1} to be $T_M(v_i)$ until we either run out of time or decide that some v_n is a sufficiently accurate approximation to M's true value function v^*. Normally we would represent each v_i as an array of real numbers indexed by the states of M; this data structure allows us to represent any possible value function exactly.

Now suppose that we wish to represent v_i, not by a lookup table, but by some other more compact data structure such as a piecewise linear function. We immediately run into two difficulties. First, computing $T_M(v_i)$ generally requires that we examine $v_i(x)$ for nearly every x in M's state space; and if M has enough states that we can't afford a lookup table, we probably can't afford to compute v_i that many times either. Second, even if we can represent v_i exactly, there is no guarantee that we can also represent $T_M(v_i)$.

To address these difficulties, we will assume that we have a sample $X_0 \subseteq S$ of states from M's state space S. X_0 should be small enough that we can examine each element repeatedly; but it should be representative enough that we can learn something about M by examining only states in X_0. Now we can define the fitted value iteration algorithm. Rather than setting v_{i+1} to $T_M(v_i)$, we will first compute $(T_M(v_i))(x)$ only for $x \in X_0$; then we will fit our piecewise linear function (or other approximator) to these training values and call the resulting function v_{i+1}.

2.1.1 Approximators as mappings

In order to reason about fitted value iteration, we will consider function approximators themselves as operators on the space of value functions. By a function approximator we mean a deterministic algorithm A which takes as input the target values for the states in X_0 and produces as output an intermediate representation which allows us to compute the fitted value at any state $x \in S$. In this definition the states in the sample X_0 are fixed, so changing X_0 results in a different function approximator.

In order to think of the algorithm A as an operator on value functions, we must reinterpret A's input and output as functions from S to $\mathbb{R}$. By doing so, we will define a mapping associated with A, $M_A : (S \mapsto \mathbb{R}) \mapsto (S \mapsto \mathbb{R})$; the input to M_A will be a function f that represents the training values for A, while the output of M_A will be another function $\hat{f} = M_A(f)$ that represents the fitted values produced by A.

If there are m states in the sample X_0, then the input to A is a vector of m real numbers. Equivalently, the input is a function f from X_0 to $\mathbb{R}$: the target value for state x is $f(x)$. Since the sample X_0 is a subset of the state space S, we can extend f to take arguments in all of S by defining $f(y)$ arbitrarily for $y \notin X_0$. This extended f is what M_A will take as input.

With this definition for f, it is easy to see how to define $\hat{f}$: for any $x \in S$, $\hat{f}(x)$ is just the fitted value at state x given the training values encoded in f. So, M_A will take the training values at states $x \in X_0$ as input (encoded as a function $f : S \to \mathbb{R}$ as described in the previous paragraph), and produce the approximate value function $\hat{f}$ as output.

In the above definition, it is important to distinguish the target function f and the learned function $\hat{f}$ from the mapping M_A: the former are real-valued functions, while the latter is a function from functions to functions. It is also important to remember that M_A is a deterministic function: since X_0 is fixed and f is deterministic, there is no element of randomness in selecting A's training data. Finally, although M_A appears to require a large amount of information as input and produce a large amount of information as output, this appearance is misleading: M_A ignores most of the information in its input, since $M_A(f)$ does not depend on $f(x)$ for $x \notin X_0$, and most of the information in $\hat{f} = M_A(f)$ is redundant, since by assumption $\hat{f}(x)$ can be computed for any x from the output of algorithm A.

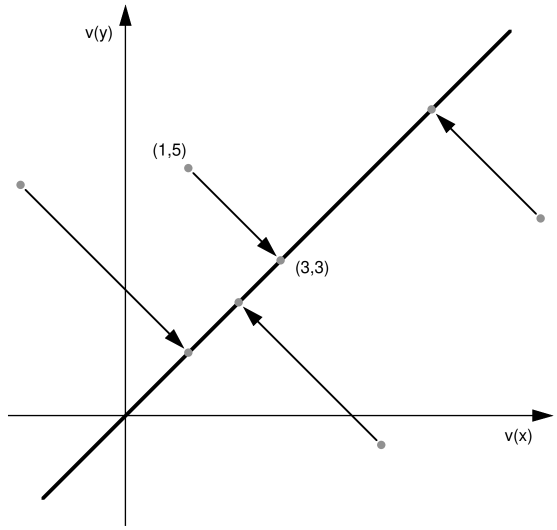


Figure 2.1: An example of the mapping for a function approximator.

Figure 2.1 shows the mapping associated with a simple function approximator. In this figure the MDP has only two states, x and y, both of which are in the sample X_0. The function approximator has a single adjustable parameter (call it a) and represents $v(x) = v(y) = a$. The algorithm for finding the parameter a from the training data $v(x)$ and $v(y)$ is $a \leftarrow (v(x) + v(y))/2$.

The set of possible value functions for a two-state MDP is equivalent to $\mathbb{R}^2$, so each point plotted in Figure 2.1 corresponds to a different possible value function. For example, the point $(1, 5)$ corresponds to the value function that has $v(x) = 1$ and $v(y) = 5$. The set of functions that the approximator can represent is shown as a thick line; these are the functions with $v(x) = v(y)$. The operator M_A maps an input (target) value function in $\mathbb{R}^2$ to an output (fitted) value function on the line $v(x) = v(y)$.

Figure 2.2 illustrates the mapping associated with a slightly more complicated function approximator. In this figure the state space of the MDP is an interval of the real line, and the sample is $X_0 = \{1, 2, 3, 4, 5\}$. The function approximator has two adjustable parameters (call them a and b) and represents the value of a state with coordinate x as $v(x) = ax + b$. The algorithm for finding a and b from the training data is linear regression.

The left column of Figure 2.2 shows two possible inputs to M_A, while the right column shows the corresponding outputs. Both the inputs and the outputs are functions from the entire state space to $\mathbb{R}$, but the input functions are plotted only at the sample points to emphasize that M_A does not depend on their value at any states $x \notin X_0$.

With our definition of M_A, we can write the fitted value iteration algorithm

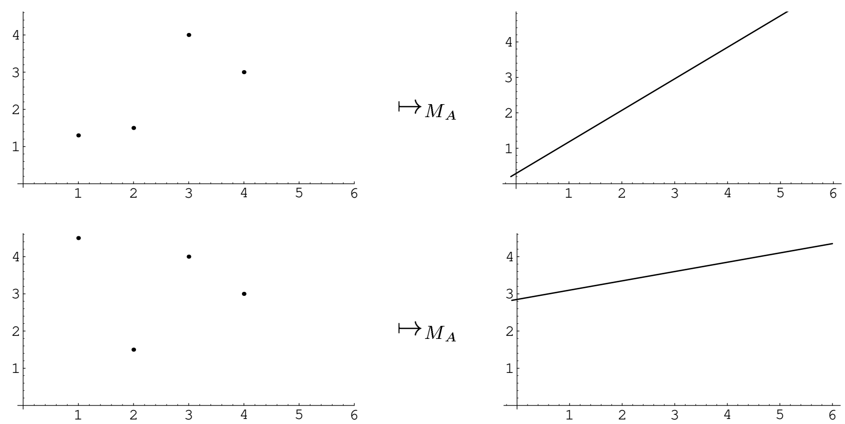


Figure 2.2: Another example of the mapping for a function approximator.

as follows. Given an initial estimate v_0 of the value function, we begin by computing $M_A(v_0)$, an approximate representation of v_0. Then we alternately apply T_M and M_A to produce the series of functions

$$v_0, M_A(v_0), T_M(M_A(v_0)), M_A(T_M(M_A(v_0))), \ldots$$

(In an actual implementation, only the functions $M_A(\ldots)$ would be represented explicitly; the functions $T_M(\ldots)$ would just be sampled at the points X_0.) Finally, when we satisfy some termination condition, we return one of the functions $M_A(\ldots)$.

The characteristics of the mapping M_A determine how it behaves when combined with value iteration. Figure 2.3 illustrates one particularly important property. As the figure shows, linear regression can exaggerate the difference between two target functions: a small difference between the target functions f and g can lead to a larger difference between the fitted functions $\hat{f}$ and $\hat{g}$. For example, while the two input functions in the left column of the figure differ by at most 1 at any state, the corresponding output functions in the right column differ by $\frac{7}{6}$ at $x = 3$. Many function approximators, such as neural nets and local weighted regression, can exaggerate this way; others, such as k-nearest-neighbor, can not.

This sort of exaggeration can cause instability in a fitted value iteration algorithm. By contrast, we will show below that approximators which never exaggerate can always be combined safely with value iteration.

More precisely, we will say that an approximator exaggerates the difference between two target functions f and g if the fitted functions $\hat{f} = M_A(f)$ and $\hat{g} = M_A(g)$ are farther apart in max norm than f and g were. Then the approximators which never exaggerate are exactly the ones whose mappings are nonexpansions in max norm: by definition, if M_A is a nonexpansion in max

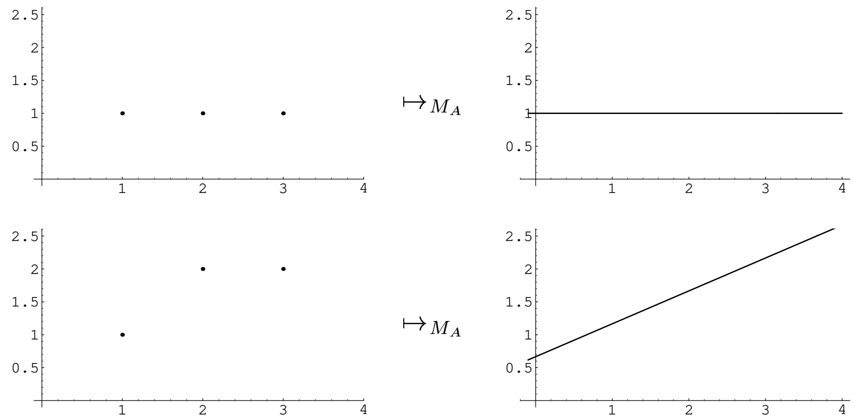


Figure 2.3: Linear regression on the sample $X_0 = \{1, 2, 3\}$.

norm, then for any target functions f and g and for any x we have

$$|\,\hat{f}(x) - \hat{g}(x)\,| \leq |\,f(x) - g(x)\,|$$

Note that we do not require that $f(x)$ and $\hat{f}(x)$ be particularly close to each other, nor that $\hat{f}(x)$ and $\hat{f}(y)$ be as close to each other as $f(x)$ and $f(y)$.

The above discussion is summarized in the following theorem:

Theorem 2.1 *Let T_M be the parallel value backup operator for some Markov decision process M with discount $\gamma < 1$. Let A be a function approximator with mapping M_A. Suppose M_A is a nonexpansion in max norm. Then $M_A \circ T_M$ has contraction factor γ; so the fitted value iteration algorithm based on A converges in max norm at the rate γ when applied to M.*

PROOF: We saw above that T_M is a contraction in max norm with factor γ. By assumption, M_A is a nonexpansion in max norm. Therefore $M_A \circ T_M$ is a contraction in max norm by the factor γ. □

One might wonder whether the converse of Theorem 2.1 is true, that is, whether the convergence of fitted value iteration with approximator A for all MDPs implies that M_A is a max-norm nonexpansion. We do not know the answer to this question, but if we add weak additional conditions on A we can prove a theorem. See Section 2.7.1.

2.1.2 Averagers

Theorem 2.1 raises the question of which function approximators can exaggerate and which can not. Unfortunately, many common approximators can. For example, as figure 2.3 demonstrates, linear regression can be an expansion in max norm; and Boyan and Moore [BM95] show that fitted value iteration with linear regression can diverge. Other methods which may diverge include standard feedforward neural nets and local weighted regression [BM95].

On the other hand, many approximation methods are nonexpansions, including local weighted averaging, k-nearest-neighbor, Bèzier patches, linear interpolation on a mesh of simplices, and multilinear interpolation on a mesh of hypercubes, as well as simpler methods like grids and other state aggregation. In fact, in addition to being nonexpansions in max norm, these methods all have two other important properties. (Not all nonexpansive function approximators have these additional properties, but many important ones do.)

First, all of the function approximation methods listed in the previous paragraph are linear in the sense that their mappings are linear functions. Linearity of the approximator in this sense does not mean that the fitted function $\hat{f}$ must be linear; instead, it means that for each x, $\hat{f}(x)$ must be a linear function of $f(x_1), f(x_2), \ldots$ for some $x_1, x_2, \ldots \in X_0$.

Second, all of these function approximation methods are monotone in the sense that their mappings are monotone functions. Again, there is no need for the fitted function $\hat{f}$ to be monotone; instead, this kind of monotonicity means that increasing any of the training values cannot decrease any of the fitted values.

We will call any function approximator that satisfies these three properties (linearity, monotonicity, and nonexpansivity) an averager. The reason for this name is that averagers are exactly the function approximators in which every fitted value $\hat{f}(x)$ is the weighted average of one or more target values $f(x_1), f(x_2), \ldots$, plus a constant offset. (The weights and offsets must be fixed, that is, they cannot depend on the target values. They can, however, depend on the choice of sample X_0, as they do in for example k-nearest-neigbor.) Averagers were first defined in [Gor95a]; the definition there is slightly less general than the one given here, but the theorems given there still hold for the more general definition. A similar class of function approximators (called interpolative representations) was defined simultaneously and independently in [TV94].

More precisely, if M has n states, then specifying an averager is equivalent to picking n real numbers k_i and n^2 nonnegative real numbers β_{ij} such that for each i we have $\sum_{j=1}^{n} \beta_{ij} \leq 1$. With these numbers, the fitted value at the ith state is defined to be

$$k_i + \sum_{j=1}^{n} \beta_{ij} f_j$$

where f_j is the target value at the jth state. The correspondence between averagers and the coefficients β_{ij} and k_i is one-to-one because, first, any linear operator M_A is specified by a unique matrix (β_{ij}) and vector (k_i); second, if any β_{ij} is negative then M_A is not monotone; and third, if $\sum_{j=1}^{n} \beta_{ij} > 1$ for any i then increasing the target value by 1 in every state will cause the fitted value at state i to increase by more than 1.

Most of the β_{ij}s will generally be zero. In particular, β_{ij} will be zero if $j \notin X_0$. In addition, β_{ij} will often be zero or near zero if states i and j are far

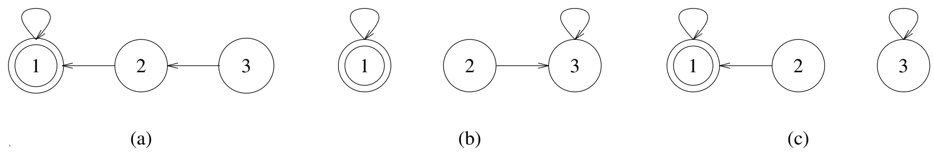


Figure 2.4: A nondiscounted deterministic Markov process and an averager. The process is shown in (a); the goal is state 1, and all arc costs are 1 except at the goal. In (b) we see the averager, represented as a Markov process: states 1 and 3 are unchanged, while $v(2)$ is replaced by $v(3)$. The embedded Markov process is shown in (c); state 3 has been disconnected, so its value estimate will diverge.

apart.

To illustrate the relationship between an averager and its coefficients we can look at a simple example. Consider a Markov decision process with five states, labelled 1 through 5. Suppose that the sample X_0 is $\{1, 5\}$, and that our averager approximates the values of states 2 through 4 by linear interpolation. Then the coefficients of this averager are $k_i = 0$ and

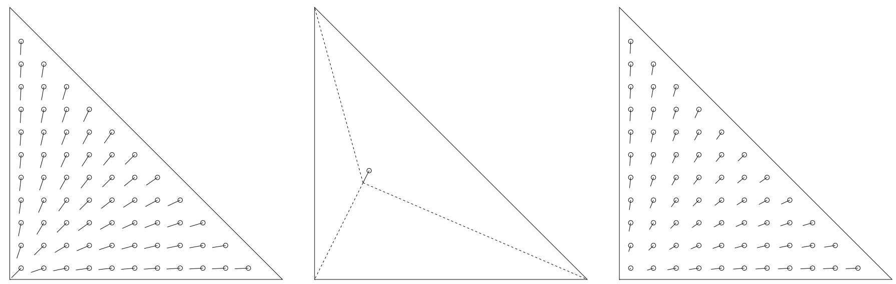
$$(\beta_{ij}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{3}{4} & 0 & 0 & 0 & \frac{1}{4} \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{4} & 0 & 0 & 0 & \frac{3}{4} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The second row of this array, for example, tells us that the fitted value for state 2 is equal to three-fourths of the target value for state 1, plus one-fourth of the target value for state 5. The fact that the middle three columns of the β matrix are zero means that states two though four are not in the sample X_0.

In this example the coefficients $\beta_{1,1}$ and $\beta_{5,5}$ are both equal to 1, which means that the fitted values at states 1 and 5 are equal to their target values. This property is not true of all averagers; for example, in k-nearest-neighbor with $k > 1$, the fitted value at a state in the sample is not equal to its target value but to the average of k different target values.

2.2 Nondiscounted processes

If $\gamma = 1$ in our MDP M, Theorem 2.1 no longer applies: $T_M \circ M_A$ is merely a nonexpansion in max norm, and so is no longer guaranteed to converge. Fortunately, there are averagers which we may use with nondiscounted MDPs. The proof relies on an intriguing relationship between averagers and fitted value iteration: we can view any averager as a Markov process, and we can view fitted value iteration as a way of combining two Markov decision processes.



(a) (b) (c)

Figure 2.5: Constructing the embedded Markov process. (a) A deterministic process: the state space is the unit triangle, and on every step the agent moves a constant distance towards the origin. The value of each state is its distance from the origin, so v^* is nonlinear. (b) A representative transition from the embedded process. For our averager, we used linear interpolation on the corners of the triangle; as before, the agent moves towards the goal, but then the averager moves it randomly to one of the corners. On average, this scattering moves the agent back away from the goal, so steps in the embedded process don't get the agent as far. The value function for the embedded process is $x + y$. (c) The expected progress the agent makes on each step.

The Markov process associated with an averager has state space S, transition matrix (β_{ij}), and cost vector (k_i). In other words, the state space is the same as M's, the probability of transitioning to state j given that the current state is i is β_{ij}, and the cost of leaving state i is k_i. (If $\sum_{j=1}^{n} \beta_{ij}$ is less than 1, we make up the difference with a transition to the terminal state $\odot$.) Since the transition matrix is (β_{ij}), there is a nonzero probability of going from i to j if and only if the fitted value at state i depends on the target value at state j. (Presumably this happens when the averager considers states i and j somehow similar.)

The reason this process is associated with the averager is that its backup operator is M_A. To see why, consider the backed up value at some state i given the starting value function v. It is equal to the cost of leaving state i, which is k_i, plus the expected value of the next state, which is $\sum_{j=1}^{n} \beta_{ij} v(j)$; in other words, it is equal to the ith component of $M_A v$. Figure 2.4(b) shows one example of a simple averager viewed as a Markov process; this averager has $\beta_{11} = \beta_{23} = \beta_{33} = 1$ and all other coefficients zero.

The simplest way to combine M with the process for the averager is to interleave their transitions, that is, to use the next-state function from M on odd time steps and the next-state function from the averager on even time steps. The result is an MDP whose next-state function depends on time. To avoid this dependence we can combine adjacent pairs of time steps, leaving an MDP whose next-state function is essentially the composition of the original two next-state functions. (We need to be careful about defining the actions of the combined

MDP: in general a combined action needs to specify an action for the first step and an entire policy for the second step. In our case, though, the second step is the Markov process for the averager, which only has one possible action. So, the actions for the combined MDP are the same as the actions for M.) It is not too hard to see that the backup operator for this new MDP is $T_M \circ M_A$, which is the same as a single step of the fitted value iteration algorithm.

As mentioned above, the state space for the new MDP is the same as the state space for M. However, since β_{ij} is zero if state j is not in the sample X_0, there will be zero probability of visiting any state outside the sample after the first time step. So, we can ignore the states in $S \setminus X_0$. In other words, the new MDP is embedded inside the old.

The embedded MDP is the same as the original one except that after every step the agent gets randomly scattered (with probabilities depending on the averager) from its current state to some nearby state in X_0. So, if a transition leads from x to y in the original MDP, and if the averager considers state $z \in X_0$ similar to y, then the same transition in the embedded MDP has a chance of moving the agent from x to z. Figure 2.4 shows a simple example of the embedded MDP; a slightly more complicated example is in Figure 2.5. As the following theorem shows (see Section 2.7.2 for a proof), exact value iteration on the embedded MDP is the same as fitted value iteration on the original MDP. A similar theorem holds for the Q-learning algorithm; see Section 2.7.4.

Theorem 2.2 (Embedded MDP) *For any averager A with mapping M_A, and for any MDP M (either discounted or nondiscounted) with parallel value backup operator T_M, the function $T_M \circ M_A$ is the parallel value backup operator for a new Markov decision process M'.*

In general, the backup operator for the embedded MDP may not be a contraction in any norm. Figure 2.4 shows an example where this backup operator diverges, since the embedded MDP has a state with infinite cost. However, we can often guarantee that the embedded MDP is well-behaved. For example, if M is discounted, or if A uses weight decay (*i.e.*, if $\sum_{j=1}^{n} \beta_{ij} < 1$ for all i), then $T_M \circ M_A$ will be a max norm contraction. Other conditions for the good behavior of the embedded MDP are discussed in [Gor95a].

2.3 Converging to what?

Until now, we have only considered the convergence or divergence of fitted dynamic programming algorithms. Of course we would like not only convergence, but convergence to a reasonable approximation of the value function.

Suppose that M is an MDP with value function v^*, and let A be an averager. What if v^* is also a fixed point of M_A? Then v^* is a fixed point of $T_M \circ M_A$; so if we can show that $T_M \circ M_A$ converges to a unique answer, we will know that it converges to the right answer. For example, if M is discounted, or if it has $E(c(x, a)) > 0$ for all $x \neq \odot$, then $T_M \circ M_A$ will converge to v^*.

If we are trying to solve a nondiscounted MDP and v^* differs slightly from the nearest fixed point of M_A, arbitrarily large errors are possible. If we are trying to solve a discounted MDP, on the other hand, we can prove a much stronger result: if we only know that the optimal value function is near a fixed point of our averager, we can guarantee an error bound for our learned value function. (A bound immediately follows (see *e.g.* [SY94]) for the loss incurred by following the corresponding greedy policy.) For a proof of the following theorem see Section 2.7.3.

Theorem 2.3 *Let v^* be the optimal value function for a finite Markov decision process M with discount factor γ. Let T_M be the parallel value backup operator for M. Let M_A be a nonexpansion in max norm. Let v^A be any fixed point of M_A. Suppose $\| v^A - v^* \| = \epsilon$, where $\| \cdot \|$ denotes max norm. Then iteration of $T_M \circ M_A$ converges to a value function v_0 so that*

$$\begin{aligned} \| v^* - v_0 \| &\leq \frac{2\gamma\epsilon}{1-\gamma} \\ \| v^* - M_A(v_0) \| &\leq 2\epsilon + \frac{2\gamma\epsilon}{1-\gamma} \end{aligned}$$

Others have derived similar bounds for smaller classes of function approximators. For example, for a bound on the error introduced by approximating a continuous MDP with a grid, see [CT89].

The sort of error bound which we have proved is particularly useful for approximators such as linear interpolation and grids which have many fixed points. Because it depends on the maximum difference between v^* and v^A, the bound is not very useful if v^* may have large discontinuities at unknown locations: if v^* has a discontinuity of height d, then any averager which can't mimic the location of this discontinuity exactly will have no representable functions (and therefore no fixed points) within $\frac{d}{2}$ of v^*.

2.4 In practice

The most common problems with approximate value iteration are oversmoothing and the introduction of barriers into the embedded MDP. By the introduction of barriers, we mean that sometimes the embedded MDP can be divided into two pieces so that the first piece contains the goal and the second piece has no transitions into the first. In this case, the estimated values of the states in the second piece will be infinite. (A special case of this situation is that, if the averager ignores the goal state, then the embedded MDP will have no transitions into the goal.) A less drastic but similar problem occurs when the second piece has only low-probability transitions to the first; in this case, the costs for states in the second piece will not be infinite, but will still be artificially inflated.

This sort of problem is likely to happen when the MDP has short transitions and when there are large regions where a single state dominates the averager.

For a particularly bad example, suppose our function approximator is 1-nearest-neighbor. If the transitions out of a sampled state x in the original MDP are shorter than half the distance to the nearest adjacent sampled state, then the only transitions out of x in the embedded MDP will lead straight back to x. So, x will have infinite cost in the embedded MDP. Similarly, in local weighted averaging with a narrow kernel, a short transition out of x in the original MDP will translate to a high probability self loop in the embedded MDP, causing x to have a finite but very large cost. In both of these examples, we can imagine that the averager is producing a drag on transitions out of x, so that actions in the embedded MDP don't get the agent as far on average as they did in the original MDP.

One way to avoid creating barriers in the embedded MDP is to make sure that no single state has the dominant weight over a large region. The best way to do so is to sample the state space more densely; but if we could afford to do that, we wouldn't need a function approximator in the first place. Another way is to increase a smoothing parameter such as kernel width or number of neighbors, and so reduce the weight of each sample point in its immediate neighborhood. Unfortunately, increasing the amount of smoothing risks oversmoothing.

Oversmoothing happens when a function approximator interacts with value iteration to wash out the features of the value function that we are interested in. In oversmoothing, the function approximator could learn a good approximation to the value function if it were trained by supervised learning, but fitted value iteraction still converges to a bad approximation. For example, if the agent must follow a long, narrow path to the goal, the scattering effect of a wide-kernel averager is almost certain to push it off of the path before it reaches the end.

Figure 2.6 demonstrates oversmoothing in a simple one-dimensional Markov process. In this process, the state space is the interval $[0, 1]$. The agent moves left a distance of .1 every time step, except when its position is already left of .1, in which case it just moves to the origin. The state $x = 0$ is terminal. The cost at state x is $.1 \cos(20\pi x)$, except that if $x < .1$ the cost is pro-rated by the distance moved. Since the period of $\cos(20\pi x)$ is equal to the distance moved on each step, the agent's cost to move a given distance remains constant throughout each trajectory and depends only on the trajectory's starting state.

The four graphs in Figure 2.6 show the performance of fitted value iteration with k-nearest-neighbor for $k = 1, 5, 10, 15$. The solid line in each graph shows the true value function $v(x) = x \cos(20\pi x)$. The dashed line shows the approximation to $v(x)$ computed by fitted value iteration with k-nearest-neighbor. For $k = 1$ this approximation is good, while for larger values of k it cuts off the peaks of $v(x)$. To demonstrate that this problem is not just due to the inherent smoothing in k-nearest-neighbor, the dotted line in each graph shows the approximation to $v(x)$ computed by supervised learning. For larger values of k it is clear that, while some of the smoothing comes from k-nearest-neighbor itself, combining k-nearest-neighbor with fitted value iteration amplifies the problem.

The reason for oversmoothing is that fitted value iteration applies the function approximator M_A over and over again to the candidate value function.

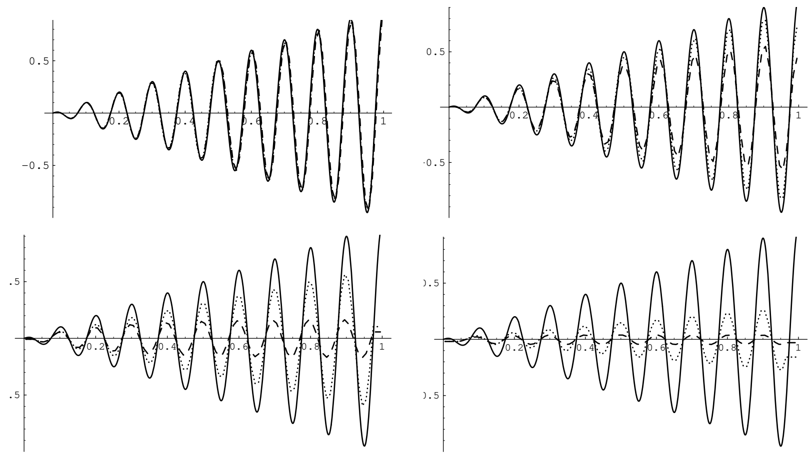


Figure 2.6: Oversmoothing in k-nearest-neighbor, for $k = 1, 5, 10, 15$ out of a sample of 200 states. The solid line is the true value function, the dashed line is its approximation with fitted value iteration and k-nearest-neighbor, and the dotted line is its approximation with supervised learning and k-nearest-neigbor.

Since M_A by definition loses some information, multiple applications of M_A may lose so much information that the resulting approximation to the value function is useless. This problem hits some function approximators harder than others: while methods like state aggregation and linear interpolation don't usually suffer too badly, methods like k-nearest neighbor with large k and local-weighted averaging with a wide kernel can have problems.

To see why fitted value iteration behaves differently with k-nearest-neighbor than with linear interpolation, consider what happens if we are lucky enough that the function approximator can represent the true value function exactly—that is, suppose $v^* = M_A v$ for some v. (The situation will be qualitatively similar if we can just represent something close to the true value function.) If we're using linear interpolation, the above assumption means that v^* is a fixed point of M_A, since reapplying linear interpolation to a linearly-interpolated function doesn't change anything. So, v^* will be a fixed point of the fitted value iteration update $T_M \circ M_A$, and we will end up with zero error. On the other hand, reapplying k-nearest-neighbor does change the result (that is, $M_A v \neq M_A M_A v$ in general), so fitted value iteration with k-nearest-neighbor can drift away from v^* and end up somewhere else.

Both of the above problems — too much smoothing and the introduction of barriers — can be reduced if we can alter our MDP so that the actions move the agent farther. For example, we might look ahead two or more time steps at each value backup. (This strategy corresponds to the dynamic programming operator $T_M^n \circ M_A$ for some $n > 1$. Since T_M^n is the backup operator for the MDP derived by composing n copies of M, the previous sections' convergence theorems also apply to $T_M^n \circ M_A$.) While in general the cost of looking ahead n steps is exponential in n, there are many circumstances where we can reduce this cost dramatically. For instance, in a physical simulation, we can choose a longer time increment; in a grid world, we can consider only the compound actions which don't contain two steps in opposite directions; and in the case of a Markov process, where there is only 1 action, the cost of lookahead is linear rather than exponential in n. (In the last case, TD(λ) [Sut88] allows us to combine lookaheads at several depths.) If actions are selected from an interval of $\mathbb{R}$, numerical minimum-finding algorithms such as Newton's method or golden section search can find a local minimum quickly. In any case, if the depth and branching factor are large enough, standard heuristic search techniques can at least chip away at the base of the exponential.

2.5 Experiments

This section describes our experiments with several Markov decision problems: two taken from [BM95], and one which shows that fitted value iteration can learn value functions in extremely high-dimensional state spaces.

2.5.1 Puddle world

In this world, the state space is the unit square, and the goal is the upper right corner. The agent has four actions, which move it up, left, right, or down by .1 unit per step. The cost of each action depends on the current state: for most states, it is the distance moved, but for states within the two "puddles," the cost is higher. See figure 2.7.

For a function approximator, we will use bilinear interpolation, defined as follows: to find the predicted value at a point (x, y), first find the corners (x_0, y_0), (x_0, y_1), (x_1, y_0), and (x_1, y_1) of the grid square containing (x, y). Interpolate along the left edge of the square between (x_0, y_0) and (x_0, y_1) to find the predicted value at (x_0, y). Similarly, interpolate along the right edge to find the predicted value at (x_1, y). Now interpolate across the square between (x_0, y) and (x_1, y) to find the predicted value at (x, y).

Figure 2.7 shows the cost function for one of the actions, the optimal value function computed on a 100×100 grid, an estimate of the optimal value function computed with bilinear interpolation on the corners of a 7×7 grid (*i.e.*, on 64 sample points), and the difference between the two estimates. Since the optimal value function is nearly piecewise linear outside the puddles, but curved inside, the interpolation performs much better outside the puddles: the root mean squared difference between the two approximations is 2.27 within one step of the puddles, and .057 elsewhere. (The lowest-resolution grid which beats bilinear interpolation's performance away from the puddles is 20×20; but even a 5×5 grid can beat its performance near the puddles.)

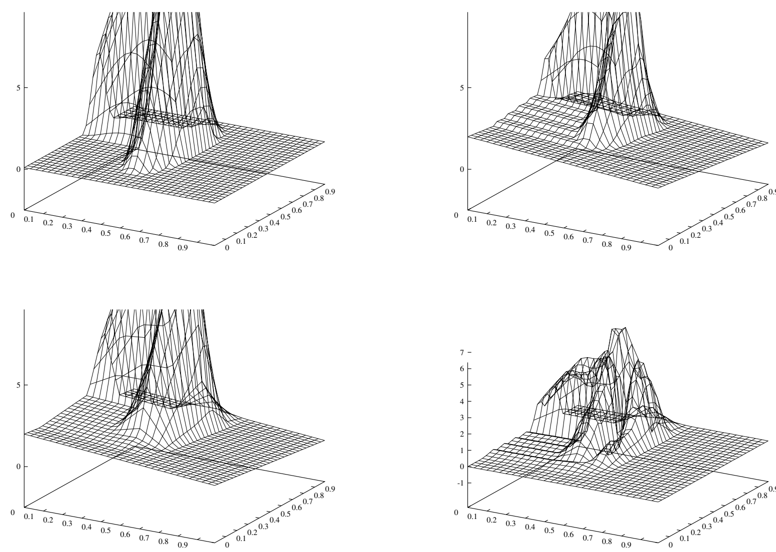


Figure 2.7: The puddle world. From top left: the cost of moving up, the optimal value function as seen by a 100×100 grid, the optimal value function as seen by bilinear interpolation on the corners of a 7×7 grid, and the second value function minus the first. Some plots are intentionally cut off at the top to preserve a constant z scale and to show detail.

2.5.2 Hill-car

In this world, the agent must drive a car up to the top of a steep hill. Unfortunately, the car's motor is weak: it can't climb the hill from a standing start. So, the agent must back the car up and get a running start. The state space is $[-1, 1] \times [-2, 2]$, which represents the position and velocity of the car; there are two actions, forward and reverse. (This formulation differs slightly from [BM95]: they allowed a third action, coast. We expect that the difference makes the problem no more or less difficult.) The cost function measures time until goal.

There are several interesting features to this world. First, the value function contains a discontinuity despite the continuous cost and transition functions: there is a sharp transition between states where the agent has just enough speed to get up the hill and those where it must back up and try again. Since most function approximators have trouble representing discontinuities, it will be instructive to examine the performance of approximate value iteration in this situation. Second, there is a long, narrow region of state space near the goal through which all optimal trajectories must pass (it is the region where the car is partway up the hill and moving quickly forward). So, excessive smoothing will cause errors over large regions of the state space. Finally, the physical simulation uses a fairly small time step, .03 seconds, so we need fine resolution in our function approximator just to make sure that we don't introduce a barrier.

The results of our experiments appear in figure 2.8. For a reference model, we fit a 128×128 grid. While this model has 16384 parameters, it is still less than perfect: the right end of the discontinuity is somewhat rough. (Boyan and Moore used a 200 by 200 grid to compute their optimal value function, and it shows no perceptible roughness at this boundary.) We also fit two smaller grids, one 64×64 and one 32×32. Finally, we fit a weighted 4-nearest neighbor model using the 1024 centers of the cells of the 32×32 grid as sample points, and another using a uniform random sample of 1000 points from the state space. Note that the nearest-neighbor methods are roughly comparable in complexity to the 32×32 grid: each one requires us to evaluate about two thousand transitions in the MDP for every value backup.

As the difference plots show, most of the error in the smaller models is concentrated around the discontinuity in the value function. Near the discontinuity, the grids perform better than the nearest-neighbor models (as we would expect, since the nearest-neighbor models tend to smooth out discontinuities). But away from the discontinuity, the nearest-neighbor models win. The 32×32 nearest-neighbor model also beats the 32×32 grid at the right end of the discontinuity: the car is moving slowly enough here that the grid thinks that one of the actions keeps the car in exactly the same place. The nearest-neighbor model, on the other hand, since it smooths more, doesn't introduce as much drag as the grid does and so doesn't have this problem. The root mean square error of the 64×64 grid (not shown) from the reference model is $0.190s$, and of the 32×32 grid is $0.336s$. The RMS error of the 4-nearest-neighbor fitter with samples at the grid points is $0.205s$. The nearest-neighbor fitter with a random

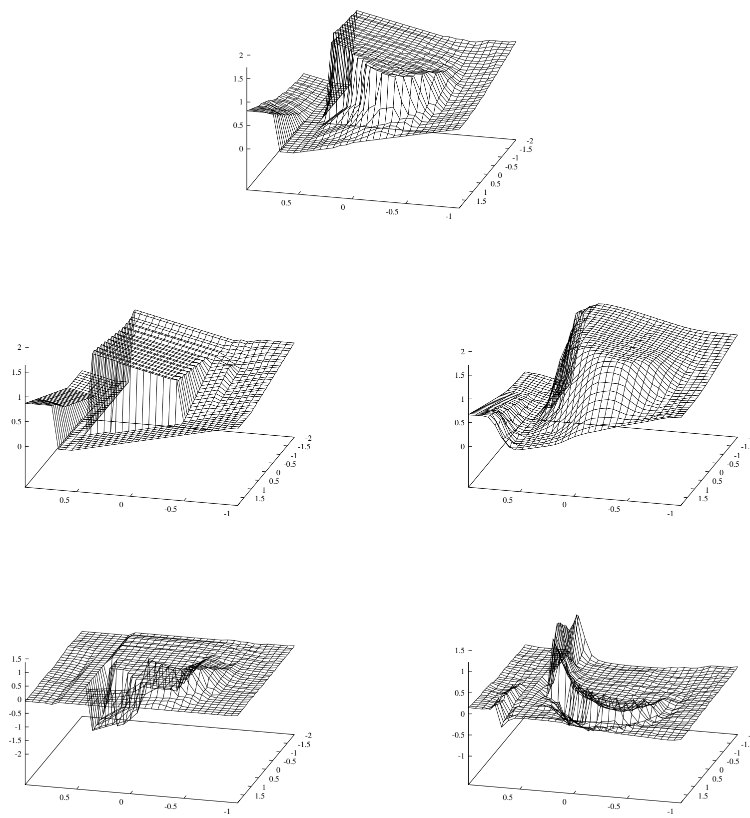


Figure 2.8: Approximations to the value function for the hill-car problem. From the top: the reference model, a 32×32 grid, a k-nearest-neighbor model, the error of the 32×32 grid, and the error of the k-nearest-neighbor model. In each plot, the x axis represents the agent's position, the y axis represents its velocity, and the z axis represents the estimated time remaining until reaching the summit at $x = .6$.

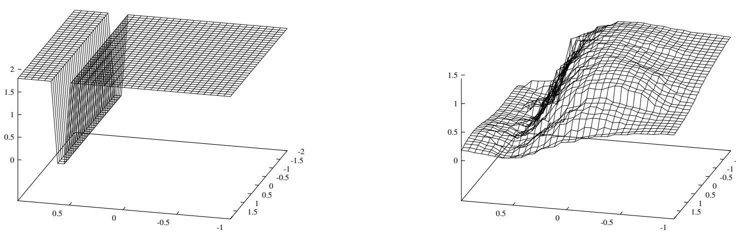


Figure 2.9: Two smaller models for the hill-car world: a divergent 12×12 grid, and a convergent nearest-neighbor model on the same 144 sample points.

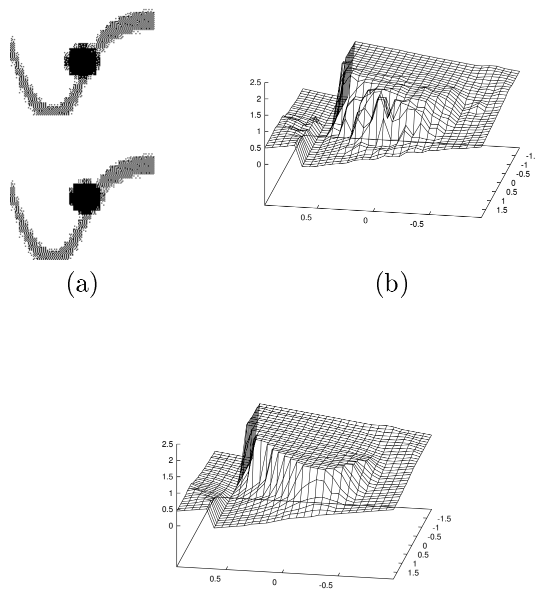
sample (not shown) performs slightly worse, but still significantly better than the 32×32 grid (one-tailed t-test gives $p = .971$): its error, averaged over 5 runs, is $0.235s$.

All of the above models are fairly large: the smallest one requires us to evaluate 2000 transitions for every value backup. Figure 2.9 shows what happens when we try to fit a smaller model. The 12×12 grid is shown after 60 iterations; it is in the process of diverging, since the transitions are too short to reach the goal from adjacent grid cells. The 4-nearest-neighbor fitter on the same 144 grid points has converged; its RMS error from the reference model is $0.278s$ (better than the 32×32 grid, despite needing to simulate fewer than one-seventh as many transitions). A 4-nearest-neighbor fitter on a random sample of size 150 (not shown) also converged, with RMS error $0.423s$.

2.5.3 Hill-car the hard way

In the previous section's formulation of this world the state space is $[-1, 1] \times [-2, 2]$, representing the position and velocity of the car. As we saw, this state space is small enough that value iteration on a reasonably-sized grid (1000 to 40000 cells, depending on the desired accuracy) can find the optimal value function. To test fitted value iteration, we expanded the state space's dimensionality a thousandfold: instead of position and velocity, we represented each state with two 32×32 grayscale pictures like the ones in figure 2.10(a), making the new state space $[0, 1]^{2048}$. The top picture shows the car's current position; the bottom one shows where it would be in $.03s$ if it took no action. A simple grid on this expanded state space is unthinkable: even if we discretized to just two gray levels per pixel, the grid would have 2^{2048} cells.

To approximate the value function, we took a random sample of 5000 legal pictures and ran fitted value iteration with local weighted averaging. In local weighted averaging, the fitted value at state x is an average of the target values at nearby sampled states x', weighted by a Gaussian kernel centered at x. We used a symmetric kernel with height 1 at the center and height $\frac{1}{e}$ when the Euclidean distance from x' to x was about 22. (We arrived at this kernel width



(a) (b)

(c)

Figure 2.10: The hill-car world.

by a coarse search: it is the narrowest kernel width we tested for which the embedded MDP was usually connected.) We repeated the experiment three times and selected the run with the median RMS error.

The resulting value function is shown in figure 2.10(b); its RMS error from the exact value function (figure 2.10(c)) is $0.155s$. By comparison, a 70×71 grid *on the original, two-dimensional problem* has RMSE $0.186s$.

2.6 Summary

In this chapter we described an algorithm called fitted value iteration, which is a generalization of state aggregation to handle any function approximator that is a nonexpansion in max norm. Such approximators include k-nearest-neighbor and linear and multilinear interpolation. We proved convergence rate and error bounds for fitted value iteration applied to a discounted Markov decision process.

To analyze fitted value iteration applied to a nondiscounted MDP, we added the additional constraints that the function approximator be linear and monotone. The resulting class of approximators, called averagers, still contains most of the popular nonexpansive approximators. We showed that running fitted value iteration with an averager on an MDP M is equivalent to running exact value iteration on a new, smaller MDP embedded within M.

Finally, we ran experiments which demonstrate that the combination of fitted value iteration with an averager can solve problems that require both pattern recognition and planning. These experiments show that fitted value iteration significantly extends the range of problems that we can solve with a provably-convergent algorithm.

2.7 Proofs

This section contains proofs that were omitted from the main text. It can be skipped without loss of continuity.

2.7.1 Can expansive approximators work?

The following theorem is almost a converse of Theorem 2.1. Instead of showing that nonexpansion of M_A is necessary to guarantee convergence for all MDPs (which would be equivalent to showing that the existence of two points x and y with $\|M_A x - M_A y\|_\infty > \|x - y\|_\infty$ is enough to find an MDP for which fitted value iteration does not converge), it requires the additional condition that $x \leq y$.

Theorem 2.4 *Suppose that the approximator A has mapping M_A, and suppose that there are two value functions $x \leq y$ such that*

$$\|M_A x - M_A y\|_\infty > \|x - y\|_\infty$$

Then there exists a Markov process M (with a finite value function) such that fitted value iteration with approximator A does not converge to a unique answer when applied to M.

PROOF: Write $M_A x = v$ and $M_A y = w$. We will construct a Markov process M such that the backup operator T_M has either $T_M v = x$ and $T_M w = y$ or $T_M w = x$ and $T_M v = y$. In the former case the fitted value iteration operator $T_M \circ M_A$ will have at least two fixed points, namely x and y, while in the latter case $T_M \circ M_A$ will have a limit cycle that alternates between x and y. In either case fitted value iteration will not converge to a unique answer.

Let s be any state where v and w differ by the maximum amount, that is, with $|v(s) - w(s)| = \|v - w\|_\infty$. We will define the process M so that every transition will end either in s or in the terminal state $\odot$. First suppose that $v(s) < w(s)$. Let i be an arbitrary state. By assumption $0 \leq y(i) - x(i) < w(s) - v(s)$. We define M's transition function so that, if i is the current state, the next state is s with probability

$$p(i) = \frac{y(i) - x(i)}{w(s) - v(s)}$$

and $\odot$ with probability $1 - p(i)$. We define M's cost function so that the cost of leaving state i is $x(i) - p(i)v(s)$. With these definitions, we can compute

$$(T_M v)(i) = p(i)v(s) + x(i) - p(i)v(s) = x(i)$$

$$(T_M w)(i) = p(i)(w(s) - v(s)) + x(i) = y(i)$$

So, we have $T_M v = x$ and $T_M w = y$.

Now suppose that $w(s) < v(s)$. In this case $0 \leq y(i) - x(i) < v(s) - w(s)$, so we can define

$$p(i) = \frac{y(i) - x(i)}{v(s) - w(s)}$$

and set the cost of leaving state i to $y(i) - p(i)v(s)$. Then

$$(T_M v)(i) = p(i)v(s) + y(i) - p(i)v(s) = y(i)$$

$$(T_M w)(i) = p(i)(w(s) - v(s)) + y(i) = x(i)$$

So, $T_M v = y$ and $T_M w = x$.

In either case, $p(i) < 1$ for all i, so M reaches the terminal state $\odot$ with probability 1 from any initial state. Therefore M's value function is finite as required. □

2.7.2 Nondiscounted case

This proof uses the definition of an averager from [Gor95a], which is slightly less general than the one given here. The proof works with only minor changes for the more general definition.

PROOF OF THEOREM 2.2: Define the embedded MDP M' as follows. It will have the same state and action spaces as M, and it will also have the same discount factor and initial distribution. We can assume without loss of generality that state 1 of M is cost-free and absorbing: if not, we can renumber the states of M starting at 2, add a new state 1 which satisfies this property, and make all of its incoming transition probabilities zero. We can also assume, again without loss of generality, that $\beta_1 = 1$ and $k_1 = 0$ (that is, that A always sets $v(1) = 0$) — again, if this property does not already hold for A, we can add a new state 1.

Suppose that, in M, action a in state x takes us to state y with probability p_{axy}. Suppose that A replaces $v(y)$ by $\beta_y k_y + \sum_z \beta_{yz} v(z)$. Then we will define the transition probabilities in M' for state x and action a to be

$$\begin{aligned} p'_{axz} &= \sum_y p_{axy}\beta_{yz} \qquad (z \neq 1) \\ p'_{ax1} &= \sum_y p_{axy}(\beta_{y1} + \beta_y) \end{aligned}$$

These transition probabilities make sense: since A is an averager, we know that $\sum_z \beta_{yz} + \beta_y$ is 1, so

$$\begin{aligned} \sum_z p'_{axz} &= \sum_{z\neq 1}\sum_y p_{axy}\beta_{yz} + \sum_y p_{axy}(\beta_{y1} + \beta_y) \\ &= \sum_y p_{axy}\left(\sum_z \beta_{yz} + \beta_y\right) \\ &= \sum_y p_{axy} = 1 \end{aligned}$$

Now suppose that, in M, performing action a from state x yields expected cost c_{xa}. Then performing action a from state x in M' yields expected cost

$$c'_{xa} = c_{xa} + \gamma \sum_{y'} p_{axy'}\beta_{y'}k_{y'}$$

Now the parallel value backup operator $T_{M'}$ for M' is

$$\begin{aligned} v(x) &\leftarrow \min_a E(c'(x,a) + \gamma v(\delta'(x,a))) \\ &= \min_a \sum_z p'_{axz}(c'_{xa} + \gamma v(z)) \\ &= \min_a \left(\sum_{z\neq 1}\left(\sum_y p_{axy}\beta_{yz}\right)(c'_{xa} + \gamma v(z)) \right. \\ &\qquad \left. + \left(\sum_y p_{axy}(\beta_{y1} + \beta_y)\right)(c'_{xa} + \gamma v(1)) \right) \end{aligned}$$

$$\begin{aligned}
&= \min_a \sum_y p_{axy} \left(\sum_{z \neq 1} \beta_{yz}(c'_{xa} + \gamma v(z)) + (\beta_{y1} + \beta_y) c'_{xa} \right) \\
&= \min_a \sum_y p_{axy} \left(c'_{xa} + \gamma \sum_{z \neq 1} \beta_{yz} v(z) \right) \\
&= \min_a \left(c'_{xa} + \gamma \sum_y p_{axy} \sum_{z \neq 1} \beta_{yz} v(z) \right) \\
&= \min_a \left(c_{xa} + \gamma \sum_{y'} p_{axy'} \beta_{y'} k_{y'} + \gamma \sum_y p_{axy} \sum_{z \neq 1} \beta_{yz} v(z) \right)
\end{aligned}$$

On the other hand, the parallel value backup operator for M is

$$\begin{aligned}
v(x) &\leftarrow \min_a E(c(x,a) + \gamma v(\delta(x,a))) \\
&= \min_a \sum_y p_{axy}(c_{xa} + \gamma v(y))
\end{aligned}$$

If we replace $v(y)$ by its approximation under A, the operator becomes $T_M \circ M_A$:

$$\begin{aligned}
v(x) &\leftarrow \min_a \sum_y p_{axy} \left(c_{xa} + \gamma(\beta_y k_y + \sum_z \beta_{yz} v(z)) \right) \\
&= \min_a \left(c_{xa} + \gamma \sum_y p_{axy} \beta_y k_y + \gamma \sum_y p_{axy} \sum_{z \neq 1} \beta_{yz} v(z) \right)
\end{aligned}$$

which is exactly the same as $T_{M'}$ above. $\square$

2.7.3 Error bounds

Proof of Theorem 2.3: By Theorem 2.1, $T_M \circ M_A$ is a contraction in max norm with factor γ, and therefore converges to some v_0. Repeated application of the triangle inequality and the definition of a contraction give

$$\begin{aligned}
\| v_0 - T_M(M_A(v^*)) \| &= \| T_M(M_A(v_0)) - T_M(M_A(v^*)) \| \\
&\leq \gamma \| v_0 - v^* \| \\
\| T_M(M_A(v^*)) - v^* \| &= \| T_M(M_A(v^*)) - T_M(v^*) \| \\
&\leq \gamma \| M_A(v^*) - v^* \| \\
&\leq \gamma \| M_A(v^*) - v^A \| + \gamma \| v^A - v^* \| \\
&= \gamma \| M_A(v^*) - M_A(v^A) \| + \gamma \| v^A - v^* \| \\
&\leq \gamma \| v^* - v^A \| + \gamma \| v^A - v^* \| \\
\| v_0 - v^* \| &\leq \| v_0 - T_M(M_A(v_*)) \| + \| T_M(M_A(v^*)) - v^* \|
\end{aligned}$$

$$
\begin{aligned}
&\le \gamma\| v_0 - v^* \| + 2\gamma \| v^* - v^A \| \\
(1-\gamma)\| v_0 - v^* \| &\le 2\gamma \| v^* - v^A \| \\
\| v_0 - v^* \| &\le \frac{2\gamma\epsilon}{1-\gamma}
\end{aligned}
$$

which is what was required. □

If we let $\gamma \to 0$, we can make the above error bound arbitrarily small. This result is somewhat counterintuitive, since A may not even be able to represent v^* exactly. The reason for this behavior is that the final step in computing v_0 is to apply T_M; when $\gamma = 0$, this step produces v^* immediately.

Approximate value iteration returns $M_A(v_0)$ rather than v_0 itself. So, an error bound for $M_A(v_0)$ would be useful. The error bound on v_0 leads directly to a bound for $M_A(v_0)$:

$$
\begin{aligned}
\| v^* - M_A(v_0) \| &\le \| v^* - v^A \| + \| v^A - M_A(v_0) \| \\
&= \epsilon + \| M_A(v^A) - M_A(v_0) \| \\
&\le \epsilon + \| v^A - v_0 \| \\
&\le \epsilon + \| v^A - v^* \| + \| v^* - v_0 \| \\
&\le 2\epsilon + \frac{2\gamma\epsilon}{1-\gamma}
\end{aligned}
$$

On the other hand, usually we use $M_A(v_0)$ by doing a one-step lookahead to find the greedy action; since this lookahead is equivalent to applying T_M again, the error bound on v_0 may be a better indicator of performance.

2.7.4 The embedded process for Q-learning

Here is the analog for Q-learning of the embedded MDP theorem. (For a definition of the Q-learning algorithm, see [Wat89].) The chief difference is that, where the theorem for value iteration considered the combined operator $T_M \circ M_A$, this version considers $M_A \circ T_M^Q$ where T_M^Q is the parallel Q-learning backup operator. The difference is necessary to keep the min operation in the Q-learning backup from getting in the way. Of course, if we show that either $T_M^Q \circ M_A$ or $M_A \circ T_M^Q$ converges from any initial guess, then the other must also converge.

This proof uses the definition of an averager from [Gor95a], which is slightly less general than the one given here. The proof works with only minor changes for the more general definition.

Theorem 2.5 (Embedded MDP for Q-learning) *For any averager A with mapping M_A, and for any MDP M (either discounted or nondiscounted) with Q-learning backup operator T_M^Q, the function $M_A \circ T_M^Q$ is the Q-learning backup operator for a new Markov decision process M'.*

PROOF: The domain of A will now be pairs of states and actions. Write β_{xayb} for the coefficient of $Q(y,b)$ in the approximation of $Q(x,a)$; write k_{xa} and β_{xa} for the constant and its coefficient.

Take an initial guess $Q(x,a)$. Write Q' for the result of applying T_M^Q to Q; write Q'' for the result of applying M_A to Q'. Then we have

$$\begin{aligned}
Q'(x,a) &= E(c(x,a) + \gamma \min_b Q(\delta(x,a),b)) \\
&= c_{xa} + \gamma \sum_y p_{xay} \min_b Q(y,b) \\
Q''(z,c) &= \sum_x \sum_a \beta_{zcxa} Q'(x,a) + \beta_{zc} k_{zc} \\
&= \sum_x \sum_a \beta_{zcxa} \left(c_{xa} + \gamma \sum_y p_{xay} \min_b Q(y,b) \right) + \beta_{zc} k_{zc} \\
&= \sum_x \sum_a \beta_{zcxa} c_{xa} + \gamma \sum_x \sum_a \beta_{zcxa} \sum_y p_{xay} \min_b Q(y,b) + \beta_{zc} k_{zc} \\
&= \left(\sum_x \sum_a \beta_{zcxa} c_{xa} + \beta_{zc} k_{zc} \right) + \\
&\quad \gamma \sum_y \left(\sum_x \sum_a \beta_{zcxa} p_{xay} \right) \min_b Q(y,b)
\end{aligned}$$

We now interpret the first parenthesis above as the cost of taking action c from state z in M'; the second parenthesis is the transition probability p'_{zcy} for M'. Note that the sum $\sum_y p'_{zcy}$ will generally be less than 1; so we will make up the difference by adding a transition in M' from state z with action c to state 1 (which is assumed as before to be cost-free and absorbing and to have $v(1) \equiv 0$). □

Chapter 3

CONVEX ANALYSIS AND INFERENCE

This chapter presents a unified framework for reasoning about worst-case regret bounds for learning algorithms. This framework is based on the theory of duality of convex functions. It brings together results from computational learning theory and Bayesian statistics, allowing us to derive new proofs of known theorems, new theorems about known algorithms, and new algorithms.

This chapter does not mention Markov decision processes explicitly. Instead, its results are at a more basic level: they treat the problem of learning without independent, identically distributed data. This problem is one of the main reasons that learning value functions for MDPs is difficult, but there are other reasons as well, so in order to use this chapter's results in a value function learning algorithm we would have to solve some additional problems.

Probably the most difficult of these problems is to decide how to score a hypothesized value function. An ideal scoring method should take as input some transitions and a value function, then decide how well the value function explains the transitions. It should take into account how likely the transitions are to have produced the observed Bellman residuals, and also how well the value function predicts which transitions are optimal choices from their starting states. Also, in order to take advantage of the results of this chapter, the score should be convex in its value-function input. This last requirement rules out such scoring methods as squared Bellman error. Chapter 4 discusses in more detail the problem of scoring value functions.

Some of the material from this chapter will appear in [Gor99].

3.1 The inference problem

We are interested in the following problem: on each time step $t = 1 \dots T$ we must choose a prediction vector w_t from a set of allowable predictions $\mathcal{W}$. Then the loss function $l_t(w)$ is revealed to us, and we are penalized $l_t(w_t)$. These penalties are additive, so our overall goal is to minimize $\sum_{t=1}^{T} l_t(w_t)$. Our choice of w_t may depend on $l_1 \dots l_{t-1}$, and possibly on some additional prior information, but it may not depend on $l_t \dots l_T$.

Many well-known inference problems, such as linear regression and estimation of mixture coefficients, are special cases of this one. To express one of these specific problems as an instance of our general inference problem, we will usually interpret the loss function l_t as encoding both a training example and a criterion to be minimized: the location of the set of minima of l_t encodes the training example, while the shape of l_t encodes the cost of deviations in each direction. This double role for l_t means that the loss function will usually change from step to step, even if we are always trying to minimize the same kind of errors. For example, if we wanted to estimate the mean of a population of numbers from a sample $z_1, z_2, \dots$, then $l_t(w)$ might be $(w - z_t)^2$. This choice of l_t encodes both the current training point z_t and the fact that we are minimizing squared error. (See Figure 3.1 for more detail.) Or, if we were interested in a linear regression of y_t on x_t, $l_t(w)$ might be $(y_t - w \cdot x_t)^2$. This choice encodes both the current example (x_t, y_t) and the fact that we want to minimize the squared prediction

Trial t	0	1	2	3	4
Prediction w_t	—	0	2	3	3
Training example z_t	—	4	5	3	8
Error type	—	Squared	Squared	Squared	Squared
Loss function $l_t(w)$	w^2	$(w-4)^2$	$(w-5)^2$	$(w-3)^2$	$(w-8)^2$
Loss of w_t	—	16	9	0	25
Ttl loss of $w_1 \ldots w_t$	0	16	25	25	50
Best constant u	4				
Loss of u	16	0	1	1	16
Ttl loss of u	16	16	17	18	34
Ttl regret	-16	0	8	7	16

Figure 3.1: An example of the MAP algorithm in action, trying to minimize sum of squared errors. The prediction at trial t is the mean of all examples up to trial $t-1$, while the comparison vector is the mean of all examples up to trial t.

error. Or, if we were trying to solve a mixture estimation problem, $l_t(w)$ might be $-\ln(w \cdot p_t)$, where w is the vector of mixture proportions and $p_{t,i}$ is the probability of the current training point under the ith model. (Here and below, the notation $p_{t,i}$ stands for the ith component of the vector p_t.) This choice of loss function encodes properties of the current example as well as the fact that we want to maximize log-likelihood.

We want to develop an algorithm for choosing a sequence of w_ts so as to minimize our total loss $\sum_{t=1}^{T} l_t(w_t)$, even if the sequence of loss functions l_t is chosen by an adversary. Unfortunately this problem is impossible without further assumptions: for example, the adversary could choose loss functions with corners or discontinuities and make the losses of two predictions v_t and w_t arbitrarily different even if v_t and w_t were close together. So, we will make two basic simplifications. The first is that we will place restrictions on the form of the functions l_t that the adversary may choose. The chief restrictions will be that l_t is convex and that a measure of the amount of information contained in l_t does not increase too quickly from trial to trial.

The second simplification is that we will seek a relative loss bound rather than an absolute one. That is, we will define a comparison class $\mathcal{U}$ of predictions, and we will seek to minimize our regret $\sum_{t=1}^{T}(l_t(w_t) - l_t(u))$ versus the best predictor $u \in \mathcal{U}$. (Often we will take $\mathcal{U} = \mathcal{W}$, so that we are comparing our predictions to the best constant prediction. Sometimes, though, we will need to take $\mathcal{U} \subset \mathcal{W}$ in order to prove a bound.) Since u can be chosen post hoc, with knowledge of the loss functions l_t, such a regret bound is a strong statement.

The focus on regret instead of just loss is the chief place where our results differ from traditional statistical estimation theory. It is what allows us to handle sequences of loss functions that are too difficult to predict: our theorems will still hold, but since there will be no comparison u that has small loss, the theorems will not tell us much about our total loss $\sum_{t=1}^{T} l_t(w_t)$.

Surprisingly, with only weak restrictions on l_t and u, we will be able to prove bounds that are similar to the best possible average-case bounds (that is, bounds where l_t is chosen by some fixed probability law). Our theorems will unify results from classical statistics (inference in exponential families and generalized linear models) with those from computational learning theory (weighted majority, aggregating algorithm, exponentiated gradient).

This regret bound framework has been studied before in [LW92, KW97, KW96, Vov90, CBFH$^+$95] among others. Also, some of our results are similar to results from classical statistics such as the Cramer-Rao variance bound [SO91]. Our theorems are more general than each of these previous results in at least one of the following ways. First, they apply to more general classes of convex loss functions, including non-differentiable ones. Second, they apply to both online (*i.e.*, bounded computation per example) and offline (unbounded computation) algorithms. Third, they apply to all sequences of loss functions, not just on average. Finally, they apply at all time steps, not just asymptotically. Our theorems are also less general than traditional statistical results in some ways. For example, while the Cramer-Rao bound requires twice-differentiability of the loss functions, it does not require global convexity, just local convexity.

All of our theorems will concern variations on the following simple and intuitively appealing algorithm, which takes as input the loss functions $l_1 \dots l_{t-1}$ observed on previous trials plus one additional loss function l_0 which encodes our prior knowledge before the first trial.

MAP ALGORITHM: Predict any $w_t \in \arg\min_w \sum_{i=0}^{t-1} l_i(w)$.

The notation $\arg\min_w f(w)$ means the set of ws that minimize f. We assume that the minimum is always achieved so that a legal prediction always exists. Conditions which ensure the existence are described below. The algorithm is called "MAP" or "maximum a posteriori" because of its Bayesian roots: if we want to apply the MAP algorithm to the problem of estimating some population parameters w from an independent identically distributed sample $z_1, z_2, \dots$, then a good choice of loss function is the negative of the log likelihood $l_t(w) = -\ln p(z_t|w)$. With this setting for l_t the MAP algorithm always chooses the prediction with maximal posterior probability given the available information. Of course, we can still use the MAP algorithm when we do not have i.i.d. samples; in this case l_t will be unrelated to any likelihood, and so "maximum a posteriori" may be a misnomer.

As the MAP algorithm is stated above it is not operational, since we may not know how to perform the required minimization. A striking feature of the MAP algorithm is that, despite the complicated machinery required to prove its theoretical properties, it often has a simple and efficient implementation. In fact, as we will see below, many well-known inference algorithms are MAP algorithms.

One example of a specific implementation of the MAP algorithm is shown in Figure 3.1. In this example, the learner is trying to minimize the sum of squared distances between its predictions w_t and a sequence of training examples

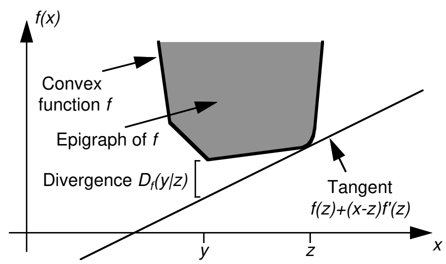


Figure 3.2: Definitions for convex functions.

$z_1 \ldots z_4$. For this problem the MAP algorithm will always predict w_t equal to the mean of all examples from trials $0 \ldots t-1$. (By convention we set $z_0 = 0$.) As shown in the figure, the best possible constant prediction is $u = 4$, since that is the mean of $z_0 \ldots z_4$. The total loss of $u = 4$ is 34, so the regret of the MAP algorithm is the difference between the loss $\sum_{t=1}^{4} l_t(w_t)$ and 34.

The rest of the paper is organized as follows. In Section 3.2 we will review some basic facts about convex analysis that we will need later on. In Section 3.3 we will outline our main results and the strategy that we will use to prove them. In Sections 3.4 and 3.5 we will prove loss bounds for the Weighted Majority algorithm, as an example of how to apply the results from Section 3.3. Section 3.6 introduces the generalized gradient descent algorithm, which is a special case of the MAP algorithm. Section 3.7 proves regret bounds for a general class of MAP algorithms that includes generalized gradient descent. Section 3.8 gives some examples of generalized gradient descent, including one which is a version of the Exponentiated Gradient algorithm. Section 3.9 treats inference in exponential families. Section 3.10 introduces generalized linear regression problems and proves regret bounds for them. Finally, Section 3.11 gives some examples of generalized linear regression algorithms, and Section 3.12 concludes.

3.2 Convex duality

For the proofs below we will need some definitions and basic results about convex functions. A convex function is any function f from a vector space $\mathcal{X}$ to $\mathbb{R} \cup \{+\infty, -\infty\}$ which satisfies

$$\lambda f(x) + (1-\lambda) f(y) \geq f(\lambda x + (1-\lambda) y)$$

for all $x, y \in \mathcal{X}$ and $\lambda \in [0,1]$. A strictly convex function is one for which we can replace $\geq$ by $>$ in the above inequality. A proper convex function is one which is always greater than $-\infty$ and not uniformly $+\infty$. The domain of f, $\operatorname{dom} f$, is the set of points where f is finite. Convex functions are continuous on $\operatorname{int}\operatorname{dom} f$, and differentiable on $\operatorname{int}\operatorname{dom} f$ except for a set of measure zero. (The notation $\operatorname{int} C$ refers to the interior of a set C, that is, the points of C which can be surrounded by an open set contained within C.)

Some special cases of convex functions are the linear functions, $f(x) = a \cdot x + b$ for a vector a and scalar b, and the indicator functions

$$\delta(x|C) = \begin{cases} 0 & x \in C \\ \infty & x \notin C \end{cases}$$

for a convex set C. (We will sometimes write a predicate instead of a set C, as in $\delta(x|\sum x_i = 1)$. There should be no danger of confusion.)

A convex function f is closed if its epigraph $\{(x,y)|y \geq f(x)\}$ is closed. The closure of f, $\operatorname{cl} f$, is the function whose epigraph is the closure of f's epigraph. For proper convex functions, closedness is the same as lower semicontinuity.

The convex hull of a function f, $\operatorname{conv} f$, is the pointwise supremum of all of the convex functions which are everywhere less than f. In other words, $\operatorname{conv} f$ is the function whose epigraph is the convex hull of f's epigraph. The convex hull always exists and is convex, although it may be the constant function $-\infty$.

The subgradient of a convex function at some point, written $\partial f(x)$, is the set of vectors a such that $f(y) \geq f(x) + (y - x) \cdot a$ for all y. In other words, the subgradient of f at x is the set of slopes of all tangent planes to f at x. We will write $\operatorname{dom} \partial f$ for the set of x such that $\partial f(x)$ is nonempty. We have

$$\operatorname{int} \operatorname{dom} f \subseteq \operatorname{dom} \partial f \subseteq \operatorname{dom} f$$

The subgradient of a smooth convex function f is single-valued on $\operatorname{int} \operatorname{dom} f$, and $\partial f(x) = \{f'(x)\}$ where $f'(x)$ stands for the usual derivative $\frac{df}{dx}$. By a slight abuse of notation we will write f' even when the subgradient is not single-valued; in this case f' will mean any (fixed) function such that $f'(x) \in \partial f(x)$. The rules for working with subgradients are similar to the rules for working with derivatives; in particular, $\partial(\lambda f)(x) = \lambda \partial f(x)$ and $\partial(f + g)(x) \supseteq \partial f(x) + \partial g(x)$. We may replace containment by equality in the latter formula under mild conditions, for example if $\operatorname{relint} \operatorname{dom} f$ and $\operatorname{relint} \operatorname{dom} g$ have a point in common.

For every function f we can define a new function f^*, called the dual of f, by the formula

$$f^*(a) = \sup_x a \cdot x - f(x)$$

The notation sup denotes the supremum or least upper bound of an expression. The dual tells us how the optimal value of a maximization problem changes if we add a linear function to the objective. The dual is always closed and convex, and $f^{**} = \operatorname{cl} \operatorname{conv} f$. If $f \geq g$ pointwise then $f^* \leq g^*$.

For example, the dual of $\exp(x)$ is $x \ln x - x$. The dual of $-\ln x$ is $-1 - \ln(-x)$. The quadratic function $x^2/2$ is self-dual. The dual of $|x|$ is $\delta(x|[-1, 1])$.

The dual of $kf(x)$ is $kf^*(\frac{x}{k})$. The dual of a linear function $a \cdot x + b$ is $\delta(x|\{a\}) - b$. The dual of $f + g$ is $f^* \,\Box\, g^*$, where the infimal convolution $u \,\Box\, v$ is defined as

$$(u \,\Box\, v)(x) = \inf_y (u(x - y) + v(y))$$

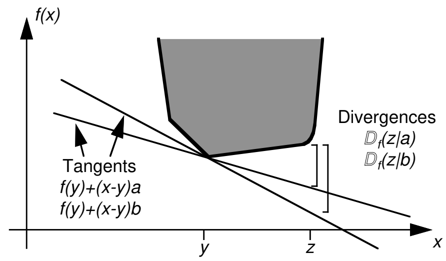


Figure 3.3: Generalized Bregman divergences.

A special case is that, if $g = a \cdot x + b$, then $(f+g)^*(x) = f^*(x-a) - b$. Another special case happens when we can partition $\mathcal{X}$ into two subspaces $\mathcal{X}_f$ and $\mathcal{X}_g$ so that $f(x)$ depends only on the component of x in $\mathcal{X}_f$ and $g(x)$ depends only on the component of x in $\mathcal{X}_g$. For example, if we write $f(x,y) = g(x) + h(y)$, then $f^*(x,y) = g^*(x) + h^*(y)$; so the dual of $|x| + |y|$ is $\delta(x|[-1,1]) + \delta(y|[-1,1])$.

The subgradients of f and f^* are (almost) inverses of each other. If f is strictly convex, then $(f^*)'(f'(x)) = x$ for all x where f' is defined. More generally, for any closed convex function f, $a \in \partial f(x)$ is equivalent to $x \in \partial f^*(a)$.

Let f be closed and convex. From the subgradient inequality, we know that

$$D_f(x|y) \stackrel{\text{def}}{=} f(x) - f(y) - (x-y) \cdot f'(y) \geq 0$$

whenever $f'(y)$ is defined. The function D_f is called a Bregman divergence. Some examples of Bregman divergences include squared Euclidean distance (which is $D_{x \cdot x}$) and information divergence (which is $D_{\sum x_i \ln x_i}$).

Bregman divergences can be either symmetric (like squared Euclidean distance) or asymmetric (like information divergence). If f is strictly convex, then $D_f(x|y) = 0$ is equivalent to $x = y$. If g is linear, then $D_{f+g} = D_f$.

The Bregman distances given by f and f^* are strongly related: if f is strictly convex, then

$$D_f(x|y) = D_{f^*}(f'(y)|f'(x))$$

If f is not strictly convex, this equality may not hold: if x is in the middle of a flat spot of f, then $f'(x)$ does not uniquely specify x.

This difficulty is a symptom of the more general problem which is illustrated in Figure 3.3: if a point $(y, f(y))$ is at a corner of f, then there are infinitely many possible tangent planes to f at y. So, there are infinitely many possible Bregman divergences all represented by $D_f(z|y)$.

One solution is to pick a divergence arbitrarily and fix D_f to mean just that divergence. This solution is the one we have been using implicitly so far, since we have defined $f'(y)$ to be an arbitrary but fixed element of $\partial f(y)$. A better solution is to generalize the definition of Bregman divergence.

We can motivate our generalization by noticing that, while a point y does not define a unique tangent plane to f, a slope a does. There is always at most one plane with slope a tangent to f, and if it exists it is given by the equation

$$f((f^*)'(a)) + (x - (f^*)'(a))a$$

There is the same ambiguity in computing $(f^*)'$ that there was in computing f', but it doesn't matter: if ∂f^* is multivalued, then each value refers to a different point along a linear segment of f, and the tangent plane at any of these points is the same.

So, we define the generalized Bregman divergence, which measures the dissimilarity between a point x and a slope a, to be

$$\mathbb{D}_f(x|a) \stackrel{\text{def}}{=} f(x) + f^*(a) - x \cdot a$$

This definition is a generalization of the original Bregman divergence since, if $a = f'(y)$, then $D_f(x|y) = \mathbb{D}_f(x|a)$. All of the properties of Bregman divergences given above carry over straightforwardly to $\mathbb{D}_f$.

Generalized Bregman divergences satisfy a simple symmetry property: our assumption that f is closed implies that

$$\mathbb{D}_f(x|a) = \mathbb{D}_{f^*}(a|x)$$

Another advantage of the new definition is that $\mathbb{D}_f(x|a)$ is defined for any x and a (although it may be infinite) and convex separately in x and in a (although it may not be convex jointly in x and a). By contrast, $D_f(x|y)$ is undefined if $\partial f(y)$ is empty, and it may not be convex in y.

A function is called positively homogeneous if $f(\lambda x) = \lambda f(x)$ for all $\lambda \geq 0$. A nonnegative, positively homogeneous, closed, convex function is called a gauge. Gauges are a generalization of norms: a norm is a gauge that is symmetric ($f(x) = f(-x)$) and strictly positive except at the origin ($f(x) = 0 \Leftrightarrow x = 0$). The dual of a gauge is an indicator function for a convex set containing the origin, and vice versa.

Two gauges g and g° are called polar to each other if

$$g^\circ(y) = \inf\{\lambda \geq 0 | (\forall x)\ x \cdot y \leq \lambda g(x)\}$$

For example, the L_p norm on $\mathbb{R}^n$ is defined to be

$$||x||_p \stackrel{\text{def}}{=} \left(\sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}}$$

and $||\cdot||_p$ and $||\cdot||_q$ are polar to each other when $\frac{1}{p} + \frac{1}{q} = 1$. Polar gauges satisfy a generalization of Hölder's inequality:

$$x \cdot y \leq g(x) g^\circ(y)$$

for all x, y, with equality iff $\lambda y \in \partial g(x)$ for some $\lambda \geq 0$. Polarity between gauges is related to duality between convex functions: if $f(x) = \frac{1}{2} g(x)^2$, then $f^*(x) = \frac{1}{2} g^\circ(x)^2$.

For more background on convex duality, see [Roc70] or [OR70].

3.3 Proof strategy

Our main result is a bound on the total regret of the MAP algorithm. It is stated below as Theorem 3.1, and an important specialization is given as Theorem 3.2. There are three basic steps in its proof and application.

Our proof is by an amortized analysis [CLR90]. So, the first step is to define a potential function for the MAP algorithm. This potential function will decrease on trials where the algorithm suffers a large regret, and increase on trials where it suffers a small or negative regret. That way, our analysis will be able to handle trials with large regret by averaging them out against other trials with smaller regret. This kind of amortized analysis is a generalization of an idea which was introduced in [LW92] and also used in many other regret-bound proofs.

The second step is to sum the regret over all trials. In order to perform this step, we will introduce some constants that, roughly speaking, summarize the amount of information available to the algorithm at the beginning of each trial. These constants depend on the type of loss function we are interested in, so we will leave them unspecified.

The third and final step is to calculate the values of the constants for the specific algorithms we wish to analyze. We will leave this step for subsequent sections.

3.3.1 Existence

Before we prove any regret bounds, we will look at when the MAP algorithm is well-defined, that is, when the minimum of $L_t = \sum_{i=0}^{t-1} l_i$ is guaranteed to be attained. While it is difficult to derive necessary and sufficient conditions for attainment of the minimum, there are some sufficient conditions which are easy to check. Throughout this section (and the rest of the paper) we will assume that each l_t is closed and convex. Because it will avoid extra notation, we will adopt the convention that any prediction is legal if L_t is the constant function $+\infty$.

The simplest sufficient condition to check is whether $\operatorname{dom} l_0^*$ is all of $\mathcal{W}$, since this condition does not depend on l_t for $t \geq 1$. Often this condition is the only one we can check. Examples of functions that satisfy this condition are $l_0(w) = w^2$ and $l_0(w) = w \ln w$. An example of a function that does not satisfy this condition is $l_0(w) = |w|$. Loosely speaking, this condition captures functions such that the norm of $l_0'(w)$ keeps increasing without bound as w approaches the border of $\operatorname{dom} l_0$.

Another simple condition to check is whether l_t attains its minimum for each t. Examples of this kind of function include $l_t(w) = (w - z)^2$ and $l_t(p) = D_{x \ln x}(p|q)$. Linear functions (such as the loss functions used in generalized gradient descent, described below) do not usually satisfy this condition.

If l_t is linear for $t \geq 1$, say $l_t(w) = w \cdot x_t$, then L_t will attain its minimum exactly when

$$X_t \stackrel{\text{def}}{=} \sum_{i=1}^{t-1} x_t \in -\operatorname{dom} \partial l_0^*$$

since this condition is true iff there is some w so that

$$\begin{aligned} 0 &\in \partial L_t(w) \\ -X_t &\in \partial l_0(w) \\ w &\in \partial l_0^*(-X_t) \end{aligned}$$

We can combine and generalize these conditions into the following lemma:

Lemma 3.1 *Suppose that the functions $l_0, l_1, \ldots$ are convex and closed. Let $m_1, m_2, \ldots$ be closed convex functions, each of which attains its minimum, such that $l_t - m_t$ is convex. Suppose there is a point ω_t for each t so that*

$$-\sum_{i=1}^{t-1} (l_i - m_i)'(\omega_t) \in \operatorname{dom} \partial l_0^*$$

Then the MAP algorithm applied to the loss functions $l_0, l_1, \ldots$ produces a legal prediction at each trial t.

PROOF: Fix a trial t and write $x_i = (l_i - m_i)'(\omega_t)$ for $1 \leq i < t$. The function $M(w) = l_0(w) + w \cdot \sum_{i=1}^{t-1} x_i$ achieves its minimum, since it is closed and convex and since the condition $0 \in \partial l_0(w) + \sum_{i=1}^{t-1} x_i$ is equivalent to $w \in \partial l_0^*(-\sum_{i=1}^{t-1} x_i)$.

The functions $l_i(w) - m_i(w) - w \cdot x_i$ also achieve their minima, since $0 \in \partial(l_i - m_i)(\omega_t) - x_i$. But L_t is the sum of M, $l_i(w) - m_i(w) - w \cdot x_i$, and $m_i(w)$ for $i = 1 \ldots t-1$. So, since the sum of closed convex minimum-achieving functions is also a closed convex minimum-achieving function, L_t achieves its minimum. $\square$

3.3.2 One-step regret

Our potential function will be a generalized Bregman divergence involving the comparison vector u, the loss functions l_t, and the MAP algorithm's current prediction w_t. The reason we use a divergence involving u and w_t is that we want to prove that, on trials where the MAP algorithm suffers a large regret compared to u, it will move its next prediction closer to u. That way, we can conclude that if it sees the same loss function again, it will incur a smaller regret.

Let $L_t = \sum_{i=0}^{t-1} l_i$, so that the w_t chosen by the MAP algorithm will be $\arg\min_w L_t(w)$. We define our potential function to be $\mathbb{D}_{L_t}(u|0)$. The potential change on each time step is given by the following lemma.

Lemma 3.2 *On trial t, the change in potential is*

$$\mathbb{D}_{L_{t+1}}(u|0) - \mathbb{D}_{L_t}(u|0) = l_t(u) - L_t^*(0) + L_{t+1}^*(0)$$

PROOF: The potential on step t is

$$\mathbb{D}_{L_t}(u|0) = L_t(u) + L_t^*(0)$$

So, the difference in potential from trial t to $t+1$ is

$$(L_{t+1} - L_t)(u) + L_{t+1}^*(0) - L_t^*(0)$$

But $L_{t+1} - L_t$ is just l_t, so the result follows. □

The function L_t^* is important, since it encodes both the best possible loss so far and the MAP algorithm's next prediction: Theorem 27.1 in [Roc70] states that $L_t^*(0) = -L_t(w_t)$ and $w_t \in \partial L_t^*(0)$. Most of the work in applying Theorem 3.1 to specific problems will come in analyzing L_t^*. For example, in the Weighted Majority proof below, $L_t^*(0)$ will be the log of the sum of the unnormalized weights, and the main part of the proof will be to connect the change in this quantity to the algorithm's loss.

3.3.3 Amortized analysis

In order to complete the proof of our bound, we need to relate the quantity $L_t^*(0) - L_{t+1}^*(0)$ to the loss of the MAP algorithm. Since the relationship depends on the type of loss function we are using, for now we will just assume that there are constants $c_1 \geq c_2 \geq \ldots > 0$ so that

$$c_t(L_t^*(0) - L_{t+1}^*(0)) \geq \tilde{l}_t(w_t) \tag{3.1}$$

Here $\tilde{l}_t$ is some function related to l_t. Often we will just use $\tilde{l}_t = l_t$, but we will sometimes need the extra generality. The smaller we take c_t, the better our bounds will be.

We can think of $1/c_t$ as a lower bound on how much information is available to the algorithm at the beginning of trial t. The best allowable value of c_t will depend on how convex L_t is when compared to l_t. For example, if every l_t is quadratic with the same second derivative, we will show below that we can take $1/c_t$ proportional to the sample size t.

With the assumption (3.1), Lemma 3.2 becomes

$$\mathbb{D}_{L_t}(u|0) - \mathbb{D}_{L_{t+1}}(u|0) \geq \frac{1}{c_t}\tilde{l}_t(w_t) - l_t(u)$$

or

$$\tilde{l}_t(w_t) \leq c_t l_t(u) + c_t \mathbb{D}_{L_t}(u|0) - c_t \mathbb{D}_{L_{t+1}}(u|0) \tag{3.2}$$

If we now apply lemma 3.2 to trial $t+1$, we get

$$\tilde{l}_{t+1}(w_{t+1}) \leq c_{t+1} l_{t+1}(u) + c_{t+1}\mathbb{D}_{L_{t+1}}(u|0) - c_{t+1}\mathbb{D}_{L_{t+2}}(u|0) \tag{3.3}$$

Notice that $\mathbb{D}_{L_{t+1}}(u|0)$ appears both in Equation 3.2 and in Equation 3.3, once with coefficient $-c_t$ and once with coefficient c_{t+1}. Since $c_{t+1} \leq c_t$ and since Bregman divergences are nonnegative, the two terms together are less than or equal to zero; so, we can drop them from our bound on total regret.

But now we have proven

Theorem 3.1 *Let $l_0, l_1, \ldots$ satisfy the assumptions of Lemma 3.1, so that the MAP algorithm produces a prediction w_t at trial t. Define $L_t = \sum_{i=1}^{t-1} l_t$. Let the constants c_t and the functions $\tilde{l}_t$ be such that $c_t(L_t^*(0) - L_{t+1}^*(0)) \geq \tilde{l}_t(w_t)$. Then the for all u total regret of the MAP algorithm is bounded by*

$$\sum_{t=1}^{T} \tilde{l}_t(w_t) \leq \sum_{t=1}^{T} c_t l_t(u) + c_1 \mathbb{D}_{l_0}(u|0)$$

PROOF: Sum lemma 3.2 over all trials, then cancel terms as described above. Finally, ignore the term containing the ending potential $\mathbb{D}_{L_{T+1}}(u|0)$. □

3.3.4 Specific bounds

All that remains is to evaluate the constants c_t for specific types of loss functions. In the following sections we will do just that. The next two sections analyze the Weighted Majority algorithm. Theorem 3.2, proved in Section 3.7, covers cases in which the one-step losses can be represented as Bregman divergences. In particular, Sections 3.6 and 3.8 cover generalized gradient descent algorithms, Section 3.9 covers inference in exponential families, and Sections 3.10 and 3.11 cover generalized linear regression algorithms including linear regression and exponentiated gradient.

3.4 Weighted Majority

One of the simplest MAP algorithms is Weighted Majority, described in [LW92]. Here we will analyze the versions which are called WMR (for randomized) and WMC (for continuous) in that paper.

WM is designed for a problem called "learning from expert advice." In this problem, the learner must choose one of N alternatives on each trial—say, which of N football games to bet a predetermined amount on. We will represent this decision with a vector w_t in the unit simplex $P = \{w \in \mathbb{R}^N | w \geq 0 \wedge \sum_i w_i = 1\}$. Picking one of the corners of the simplex means betting on the corresponding game. Picking a vector in the middle means either choosing a game to bet on at random (with probabilities w_t) or splitting the bet among the games (with proportions w_t). These two interpretations yield the WMR and WMC algorithms respectively. Since this is the only difference between WMR and WMC, we will analyze both algorithms together and use WM to refer to either one.

In either WMR or WMC, the learner then finds out which bets paid off and receives a loss of $w_t \cdot x_t$, where $x_{t,i}$ is the loss for betting on the ith game. (In WMC, the loss is deterministic, while in WMR, $w_t \cdot x_t$ is the expected loss. The expectation is over the learner's randomization.) For notational convenience, we will assume that $0 \leq x_{t,i} \leq 1$. We assume that the learner has no outside information beyond the history of losses.

The above description of the expert advice problem is a little more general than the version in [LW92]. That paper assumes that the learner is trying to solve a classification problem. There are N experts who claim to know the answer. The ith decision corresponds to agreeing with the ith expert, and $x_{t,i}$ is the prediction error of the ith expert. This version of learning from expert advice is a simple example of a regression problem (see below).

To solve the expert advice problem, WM follows a simple strategy. Whenever an expert makes a mistake (*i.e.*, has a loss of 1), WM reduces that expert's weight by a constant factor $\beta \in (0,1)$, then renormalizes to keep the sum of the weights equal to 1. Experts with losses less than 1 have their weights reduced less. More specifically, define $X_t = \sum_{i=1}^{t-1} x_t$. Write $v_{t,i} = \beta^{X_{t,i}}$. Let $Z_t = 1/\sum_i v_{t,i}$. Then WM predicts $w_t = Z_t v_t$. (Actually, [LW92] allows some flexibility in choosing X_t, but this is one of the allowed choices.)

To design a MAP algorithm for learning from expert advice, we just need to pick a prior loss function l_0, since we already know $l_t(w) = w \cdot x_t$ for $t \geq 1$. In order to make sure that our predictions are always in the unit simplex P, we will set $l_0(w) = \infty$ for $w \notin P$. A reasonable choice of l_0 for $w \in P$ is some multiple of the entropy function, making

$$l_0(w) \propto H(w)$$

$$H(w) \stackrel{\text{def}}{=} \delta(w|P) + \sum_i w_i \ln w_i \tag{3.4}$$

It is easy to verify that

$$H^*(x) = \ln \sum_i \exp(x_i)$$

$$\frac{d}{dx_i} H^*(x) = \exp(x_i) / \sum_j \exp(x_j)$$

To duplicate WM, we will pick $l_0 = \frac{1}{-\ln\beta} H(w)$. (This choice of l_0 means that w_1 will be at the center of P; it is easy to accomodate other starting vectors by adding a linear function to l_0 to move its minimum to the desired w_1.) Then

$$l_0^*(x) = \frac{1}{-\ln\beta} H^*(-x \ln\beta)$$

$$(l_0^*)'(x) = (H^*)'(-x\ln\beta)$$

Furthermore, since $L_t(w) = l_0(w) + X_t \cdot w$, we have $L_t^*(x) = l_0^*(x - X_t)$. That means that our prediction on step t will be $w_t = (l_0^*)'(-X_t) = (H^*)'(X_t \ln\beta)$, or

$$w_{t,i} = v_{t,i} / \sum_j v_{t,j}$$

$$v_{t,i} = \beta^{X_{t,i}}$$

which is identical to the prediction of WM.

Now that we have expressed WM as a MAP algorithm, we can analyze it by applying Theorem 3.1. To do so, we must compute the constants c_t. It is easy to see that taking $c_t = -\ln\beta/(1-\beta)$ for all t satisfies the assumptions of Theorem 3.1, since we can write

$$\begin{aligned}
-(L^*_{t+1}(0) - L^*_t(0))\ln\beta &= H^*(X_{t+1}\ln\beta) - H^*(X_t\ln\beta) \\
&= \ln\frac{\sum_i \beta^{X_{t,i}+x_{t,i}}}{\sum_i \beta^{X_{t,i}}} \\
&= \ln\sum_i \beta^{x_{t,i}} w_{t,i} \\
&\leq \ln\sum_i (1-(1-\beta)x_{t,i})w_{t,i} \\
&= \ln(1-(1-\beta)x_t \cdot w_t) \\
&\leq -(1-\beta)x_t \cdot w_t
\end{aligned}$$

The first inequality holds because $\beta^x \leq 1-(1-\beta)x$ for $\beta > 0$ and $x \in [0,1]$, while the second holds because $\ln(1-x) \leq -x$. So now we have proven

Corollary 3.1 *The loss of WM with parameter β is bounded by*

$$\sum_{t=1}^{T} x_t \cdot w_t \leq \frac{-\ln\beta}{1-\beta}\sum_{t=1}^{T} x_t \cdot u + \frac{1}{1-\beta}\mathbb{D}_H(u|0)$$

PROOF: Apply Theorem 3.1 with $\tilde{l}_t = l_t$ and $c_t = -\ln\beta/(1-\beta)$. Then replace l_0 by $\frac{1}{-\ln\beta}H$. □

If we now note that $\mathbb{D}_H(u|0) \leq \ln N$ for all $u \in P$, the above result is identical to Corollary 6.1 in [LW92].

3.5 Log loss

In step t of Weighted Majority the learner is charged the loss $x_t \cdot w_t$, where $x_{t,i}$ is the loss of the ith expert. For some problems it may be more appropriate to use the loss function $l_t(w) = -\ln(y_t \cdot w)$ for some vector y_t instead. Two examples are the portfolio selection problem and the mixture estimation problem.

In the portfolio selection problem, the learner is presented with N investments on each time step. After the learner chooses what fraction of its fortune to invest in each alternative, investment i grows by a factor of $y_{t,i}$. So, if the learner puts a fraction $w_{t,i}$ in each investment, its total wealth grows by a factor of $w_t \cdot y_t$. Since, in our framework, we combine losses from different trials by adding them, we need to take the log of the wealth changes. That way the total of the log wealth changes will be the log of the total wealth change. Since losses are the negative of gains that leaves us with the penalty $-\ln(w_t \cdot y_t)$.

In the mixture estimation problem, the learner must discover the coefficients in a mixture of N probability distributions. After choosing mixture coefficients

w_t, the learner receives a new training example and computes the probability $y_{t,i}$ assigned by the ith probability distribution to the new example. Since we want to maximize likelihood, or equivalently minimize negative log likelihood, we charge the learner a loss of $-\ln(w_t \cdot y_t)$.

If we write $x_{t,i} = -\ln y_{t,i}$ we can run WM with the vectors x_t. In other words, we can compute w_t according to the equations

$$w_{t,i} = v_{t,i} / \sum_j v_{t,j}$$

$$v_{t,i} = \beta^{X_{t,i}}$$

$$X_t = \sum_{i=1}^{t-1} x_t$$

We will call the resulting algorithm WM-log, even though it has the exact same series of computational steps as WM, to emphasize that we want to prove bounds on its log loss $\sum_t \ln(w_t \cdot y_t)$. Just as before, we will assume that $x_{t,i} \in [0,1]$ for notational convenience. It turns out that WM-log is a special case of the Aggregating Algorithm of [Vov90].

The WM-log update has a particularly simple interpretation in the portfolio selection problem. If we let $\eta = 1$ (so that $\beta = 1/e$), then the fraction of money in the ith investment at step t is $\exp(-X_{t,i}) / \sum_i \exp(-X_{t,i})$. The rate of growth on step t is $w_t \cdot y_t = \sum_i \exp(-X_{t+1,i}) / \sum_i \exp(-X_{t,i})$, so we can prove by induction that a fortune of $\$N$ on step 1 grows to a fortune of $\sum_i \exp(-X_{t,i})$ dollars on step t. So, the amount of money in the ith investment on step t is just $\exp(-X_{t,i}) = \prod_t y_{t,i}$ dollars. But this is exactly the amount which would be in the ith investment if we had just invested \$1 in each investment on step 1 and let it sit. And, in fact, the bound which we will prove below is equivalent to the obvious observation that investing \$1 in each investment earns at least $1/N$ times as much as investing $\$N$ in the best investment.

To analyze the WM-log algorithm we will compare our performance, not to the best vector $u \in P$, but only to the best individual expert (*i.e.*, the best corner of P). Write $\tilde{P}$ for the set of corners of P. Then if we write $m_t(w) = w \cdot x_t$,

$$l_t(w) - l_t(u) \leq m_t(w) - m_t(u) \qquad \forall u \in \tilde{P}, \forall w \in P$$

since m_t touches l_t at each corner of P but lies above l_t elsewhere in P.

If we now run the MAP algorithm with loss functions $H, m_1, m_2, \ldots$, then the analysis of the Section 3.4 shows that our predictions will be identical to WM-log with learning rate $\eta = 1$. Furthermore, with $L_t = H + \sum_{i=1}^{t-1} m_t$, we have

$$L^*_{t+1}(0) - L^*_t(0) = \ln \sum_i \exp(-x_{t,i}) w_{t,i} = \ln y_t \cdot w_t = -l_t(w_t)$$

So, we can apply Theorem 3.1 to the loss functions $H, m_1, m_2, \ldots$ with $c_t = 1$ and $\tilde{l}_t = l_t$. The result is that

$$\sum_{t=1}^{T} m_t(w_t) \leq \sum_{t=1}^{T} l_t(u) + \mathbb{D}_H(u|0) \qquad \forall u \in \tilde{P}$$

With the substitutions $m_t(w_t) = l_t(w_t)$ and $\mathbb{D}_H(u|0) = \ln N$, this becomes

$$\sum_{t=1}^{T} l_t(w_t) \leq \sum_{t=1}^{T} l_t(u) + \ln N \qquad \forall u \in \tilde{P}$$

This bound is equivalent to Equation (3.4) in [HKW98]. (That equation refers to a constant c_L, which plays the same role there that $1/\eta$ does here, and which is set to 1 for the analysis of the WM-log algorithm.)

So, for the WM and WM-log algorithms, our regret bounds are the same as the bounds previously obtained in the literature. As we would hope for a general framework for regret bounds, once we set up WM and WM-log as MAP algorithms, their proofs are similar: we evaluate the constant $c = c_t$ and apply Theorem 3.1. We can follow a similar strategy for the other MAP algorithms described below. Since some of these proofs are more complicated, we will collect some of the overlap into Theorems 3.2 and 3.3.

3.6 Generalized gradient descent

In the previous two sections we analyzed simple MAP algorithms in which all of the loss functions except the prior were linear. In the first, the loss functions started out linear, while in the second, we bounded the true loss functions by a linear approximation. Because of the linearity of the loss functions, it was easy to compute the prediction w_t on each time step: the update rules for WM and WM-log are both of the form $w_t = f(-\eta X_t)$, where X_t is the sum of the gradients of the previous loss functions and f is a function that we can compute efficiently.

We would like to be able to play the same trick for an arbitrary convex loss function l_t. That is, we would like to bound l_t by a linear function m_t, then apply the MAP algorithm to the functions m_t instead of l_t so that it will run more efficiently. Of course, the predictions will be different if we use m_t in place of l_t, and so the regret may be larger. But, we may have to do significantly less work per trial, and we will still be able to bound the regret.

The key inequalities which allowed us to replace l_t by m_t in the previous section were

$$\begin{aligned} l_t(w_t) &\leq m_t(w_t) \\ l_t(u) &\geq m_t(u) \qquad \forall u \in \mathcal{U} \end{aligned} \tag{3.5}$$

If $\mathcal{U} = \mathcal{W}$ then these inequalities force m_t to be tangent to l_t at w_t; if $\mathcal{U} \subset \mathcal{W}$ then m_t may be a secant to l_t that passes above $(w_t, l_t(w_t))$. Subtracting the second

Name	Link $(l_0^*)'$	Loss l_0
Identity	a	$\frac{1}{2}w^2$
Logistic	$\frac{1}{1+\exp(-a)}$	$w \ln w + (1-w)\ln(1-w)$
Inverse logistic	$\ln \frac{a}{1-a}$	$\ln(1+\exp(w))$
Exponential	$\exp a$	$w \ln w - w$
Logarithmic	$\ln a$	$\exp w$
Normalized exponential	$\frac{\exp a_i}{\sum_i \exp a_i}$	$\sum_i w_i \ln w_i - 1 + \delta(w \mid \sum_i w_i = 1)$

Figure 3.4: Some examples of link functions.

inequality from the first gives $l_t(w_t) - l_t(u) \leq m_t(w_t) - m_t(u)$ for all $u \in \mathcal{U}$, so that when we apply Theorem 3.1 to bound the difference $m_t(w_t) - m_t(u)$ we also get a bound on the regret $l_t(w_t) - l_t(u)$.

In the previous section we achieved Equation 3.5 by restricting $\mathcal{U}$ to the corners of the unit simplex, even though w_t was allowed to range over the entire simplex. In general we want to set $\mathcal{U}$ to the range of w_t, and in this case the only suitable linear functions m_t are those which are tangent to l_t at w_t.

If we set m_t to be a tangent to l_t at w_t, $m_t(w) = l_t(w_t) + (w - w_t) \cdot l_t'(w_t)$, and then feed the sequence of loss functions $l_0, m_1, m_2, \ldots$ to the MAP algorithm, the result is an algorithm called generalized gradient descent or GGD. It is "generalized" because, when l_0 is quadratic, the update rule reduces to ordinary gradient descent. We can write the GGD update rule as follows:

$$\text{GGD ALGORITHM: Predict } w_t \in \arg\min_w \left[l_0(w) + w \cdot \sum_{i=1}^{t-1} l_i'(w_i) \right].$$

The GGD algorithm is often written in an additive form that looks different from its statement above. If we write $X_t = \sum_{i=1}^{t-1} l_i'(w_i)$ then the additive form of the GGD prediction rule is $w_t = f(-\eta X_t)$. Here η is a learning rate and f is a function from $\mathbb{R}^n$ to $\mathbb{R}^n$ satisfying appropriate conditions. For example, choosing f to be the identity yields ordinary gradient descent. The advantage of this form of the prediction rule comes from the fact it may be difficult to compute l_0 from f, while it is often easier to compute f from l_0; so, if we are given f, we can use the additive form of the GGD rule without needing to compute l_0.

We can prove that the two forms of the GGD algorithm are equivalent: if $\eta = 1$, then we can set $f = (l_0^*)'$. For different learning rates we can just multiply l_0 by a constant, since $(\frac{1}{\eta} l_0)^*(x) = \frac{1}{\eta} l_0^*(\eta x)$ and so $((\frac{1}{\eta} l_0)^*)'(x) = (l_0^*)'(\eta x)$.

The function $f(x)$ (or equivalently $(l_0^*)'(\frac{x}{\eta})$) is called a link function. Figure 3.4 shows some useful link functions and their corresponding loss functions. The one-dimensional link functions in Figure 3.4 can easily be generalized to multiple dimensions by applying them separately to each coordinate.

Some examples of GGD algorithms are ordinary gradient descent, the perceptron learning rule, and the Exponentiated Gradient algorithm of [KW97].

We will examine some of these algorithms in more detail below. But first, we will prove regret bounds for a class of algorithms that includes GGD.

3.7 General regret bounds

3.7.1 Preliminaries

In many common MAP algorithms, each individual loss function can be written as a Bregman divergence. For example, in linear regression, the loss functions are of the form $(y_t - w_t \cdot x_t)^2$, which we may think of as a scaled Euclidean distance between w_t and any of the infinitely many perfect predictions w satisfying $y_t = w \cdot x_t$. (The scaling is such that all directions perpendicular to x_t have weight zero.) For a more general example, in GGD, if we adopt the convention that $\inf_w l_t(w) = 0$, then the loss $m_t(w_t)$ is $l_t(w_t) = \mathbb{D}_{l_t}(w_t|0)$. Or, for another example, in inference of the natural parameter in an exponential family, we will see below that the appropriate loss function is $\mathbb{D}_l(w_t|a_t)$ for a fixed l. In this section we will derive regret bounds that hold when the loss functions are divergences.

To that end, assume that we are running the MAP algorithm with loss functions $l_0, m_1, m_2, \ldots$, and that $m_t(w_t) = \mathbb{D}_{l_t}(w_t|a_t)$. Also assume $m_t(w) \leq \mathbb{D}_{l_t}(w|a_t)$ for all w. (These inequalities are a tangency condition similar to (3.5).) Write $L_t = l_0 + \sum_{i=1}^{t-1} m_t$. This notation is similar to the notation from the section on GGD, but in this section we are not assuming that the functions m_t are linear. In particular, we may take $m_t = l_t$.

In order to bound the loss of the MAP algorithm, we have to make sure that the prior loss L_t before each trial t is sufficiently convex. To see why, consider what would happen if we took $l_0 = L_1$ to be $\frac{1}{\eta}\delta(w|[0,1])$. With this choice of prior loss, our predicted w can change discontinuously from 0 to 1 even when the one-step loss has only a small gradient. So, for example, if we see $m_1 = w/2$ and then $m_2, m_3, \ldots = (1-w), w, (1-w), w, \ldots$, our predictions will alternate between 0 and 1 no matter how small η is. In fact, we will always choose the worst possible w, and so our loss will be twice that of the comparison vector $u = .5$.

We also have to make sure that the one-step divergence functions l_t for $t \geq 1$ are not too convex. If they are, we can cause the MAP algorithm to suffer an arbitrarily large regret per trial: the more convex l_t is as compared to L_t, the more of an advantage it is to pick the comparison vector after knowing m_t. For example, if $l_0(w) = w^2$ (so that $w_1 = 0$), then the loss function $m_1(w) = 10^6(w-1)^2$ will cause the MAP algorithm a loss of 10^6, while the optimal comparison vector $\frac{1}{1+10^{-6}}$ will suffer a loss of approximately 10^{-6} even though its l_0-divergence from w_1 is less than 1.

So, to ensure that L_t is sufficiently convex, we will pick a gauge g and constants $\eta_t \in (0,1)$ and require that

$$\eta_t \mathbb{D}_{L_t}(v|a) \geq \frac{1}{2}(g(v-w))^2$$

for all v and w and $a \in \partial L_t(w)$. And, to ensure that l_t is not too convex, we will require that

$$\mathbb{D}_{l_t^*}(a_t|w) \geq \frac{1}{2}(g^\circ(a_t - a))^2$$

for all w and $a \in \partial l_t(w)$.

A consequence of the first assumption is that

$$L_t(w) - L_t(w_t) \geq \frac{1}{2}(g(w - w_t))^2$$

since the LHS is equal to $\mathbb{D}_{L_t}(w|0)$ and $0 \in \partial L_t(w_t)$. A consequence of the second assumption is that

$$m_t(w_t) \geq \frac{1}{2}(g^\circ(-m_t'(w_t)))^2$$

as long as $\partial m_t(w_t)$ is nonempty, since

$$\begin{aligned} m_t(w_t) &= l_t(w_t) \\ &= \mathbb{D}_{l_t}(w_t|a_t) \\ &\geq \frac{1}{2}(g^\circ(a_t - a))^2 \end{aligned}$$

for any $a \in \partial l_t(w_t)$, and since $\partial l_t(w_t) - a_t \supseteq \partial m_t(w_t)$.

Scaling the gauge g will scale η_t inversely. So, in order to make the constant η_t as small as possible in the first assumption, it is important to take g to be as shallow as possible while still satisfying the second assumption.

3.7.2 Examples

To interpret our assumptions, it will help to compute the best gauge g and learning rate η for some examples. First suppose that L_t and l_t are both quadratic, say $L_t(w) = \frac{k}{2}w^{\mathrm{T}}Mw$ and $l_t(w) = \frac{1}{2}w^{\mathrm{T}}Mw$ for some symmetric positive definite matrix M. (This choice of l_t means that $m_t(w_t) = \frac{1}{2}(w_t - z_t)^{\mathrm{T}}M(w_t - z_t)$, where $z_t = M^{-1}a_t$.) Then we can choose $\eta_t = \frac{1}{k}$ and $g(w) = \sqrt{w^{\mathrm{T}}Mw}$, since

$$\frac{1}{k}\mathbb{D}_{L_t}(v|a) = \frac{1}{2}(v - w)^{\mathrm{T}}M(v - w) = \frac{1}{2}g(v - w)^2$$

where $w = (kM)^{-1}a$, and

$$\mathbb{D}_{l_t^*}(a_t|w) = \frac{1}{2}a_t^{\mathrm{T}}M^{-1}a_t + \frac{1}{2}w^{\mathrm{T}}Mw - a_t \cdot w = \frac{1}{2}g^\circ(a_t - a)^2$$

where $a = Mw$.

Or suppose that l_t is quadratic but L_t is proportional to the entropy function H defined in Equation 3.4. In particular, let

$$l_t(w) = \frac{1}{2}||w||_2^2$$

$$L_t(w) = k\mathbb{D}_H(w|0)$$

It is well known that $D_H(v|w) \geq 2||v-w||_2^2$ for any v, w. So, $\frac{1}{4k}D_{L_t}(v|w) \geq \frac{1}{2}||v-w||_2^2$. And just as in the previous example $\mathbb{D}_{l_t^*}(a_t|w) \geq \frac{1}{2}||a_t - w||_2^2$. So, we can choose g to be Euclidean distance and let $\eta_t = \frac{1}{4k}$.

These two examples show that g and η together provide a global analog to the Fisher information matrix. When the Fisher information $L_t''(w)$ is constant over all possible parameter values w, as it is in the first example, the local and global information measures are the same. On the other hand, when the Fisher information varies, as it does in the second example, the global measure may be much more conservative. This conservatism is necessary: in the average case we can count on having our estimates stay near the optimal value, while in the worst case our opponent can cause our estimates to wander into a region with lower information.

Finally, suppose that $l_t(w) = \frac{1}{2}(y_t - w \cdot x_t)^2$, and let $a_t = 0$ so that $m_t(w_t) = l_t(w_t)$. This choice of loss function is appropriate for linear regression problems. It depends on w only through $w \cdot x_t$, so any change in w perpendicular to x_t leaves l_t constant. That means that we can represent l_t^* as the sum of two components, one of which depends only on $w \cdot x_t$ and the other of which depends only on $w \setminus x_t \stackrel{\text{def}}{=} w - \frac{w \cdot x_t}{x_t \cdot x_t} x_t$. A little algebra shows

$$l_t^*(x) = \delta(x | x \setminus x_t = 0) + \frac{x \cdot x_t}{x_t \cdot x_t} y + \frac{1}{2}\left(\frac{x \cdot x_t}{x_t \cdot x_t}\right)^2$$

In other words, l_t^* is infinite everywhere except along the line through x_t, and along that line it is quadratic. The quadratic term (the last term in the expression above) is scaled so that it is equal to $\frac{1}{2}$ at x_t and $-x_t$. So, to bound l_t^*, we will need to make some assumption about x_t.

If we suppose that the gauge g is symmetric and scaled so that $g^\circ(x_t) \leq 1$, then it is not hard to see that $\mathbb{D}_{l_t^*}(0|w) = l_t(w) \geq \frac{1}{2}(g^\circ(x))^2$, since the latter expression is also quadratic along the line through x_t and scaled so that it is no larger than $\frac{1}{2}$ at $\pm x_t$. So, for example, if $||x_t||_\infty \leq X$, we can take $g(w)$ to be $X||w||_1$.

Now, since $||w||_1 \leq ||w||_2$, we have $D_H(v|w) \geq 2||v-w||_1^2$. So, if $L_t(w) = k\mathbb{D}_H(w|0)$, we can take $\eta_t = \frac{X^2}{4k}$.

3.7.3 The bound

We will now prove our regret bound.

Theorem 3.2 *Suppose that the loss functions $l_0, m_1, m_2, \ldots$ satisfy the conditions of Lemma 3.1, so that the MAP algorithm applied to these loss functions always produces a prediction w_t at each trial. Suppose that for all t, $\partial m_t(w_t)$ is nonempty, $m_t(w_t) = \mathbb{D}_{l_t}(w_t|a_t)$, and $m_t(w) \leq \mathbb{D}_{l_t}(w|a_t)$ for all w. Write $L_t = l_0 + \sum_{i=1}^{t-1} m_t$. Suppose that there exists a gauge g and constants*

$1 > \eta_1 \geq \eta_2 \geq \ldots > 0$ so that for all t we have

$$\eta_t \mathbb{D}_{L_t}(v|a) \geq \frac{1}{2}(g(v-w))^2$$

for all v and w and $a \in \partial L_t(w)$ and

$$\mathbb{D}_{l_t^*}(a_t|w) \geq \frac{1}{2}(g^\circ(a_t - a))^2$$

for all w and $a \in \partial l_t(w)$. Then the loss of the MAP algorithm is bounded by

$$\sum_{t=1}^{T} m_t(w_t) \leq \sum_{t=1}^{T} \frac{1}{1-\eta_t} m_t(u) + \frac{1}{1-\eta_1} \mathbb{D}_{l_0}(u|0)$$

Proof: We have

$$\begin{aligned}
L_t(w) &\geq \frac{1}{2\eta_t}(g(w-w_t))^2 + L_t(w_t) \\
m_t(w) &\geq m_t(w_t) + (w-w_t)\cdot m_t'(w_t) \\
L_{t+1}(w) &\geq \frac{1}{2\eta_t}(g(w-w_t))^2 + (w-w_t)\cdot m_t'(w_t) + L_t(w_t) + m_t(w_t) \\
L_{t+1}^*(x) &\leq \frac{1}{2\eta_t}(g^\circ(\eta_t(x - m_t'(w_t))))^2 + x\cdot w_t - L_t(w_t) - m_t(w_t) \\
L_{t+1}^*(0) - L_t^*(0) &\leq \frac{1}{2\eta_t}(g^\circ(-\eta_t m_t'(w_t)))^2 - m_t(w_t) \\
&\leq (\eta_t - 1) m_t(w_t)
\end{aligned}$$

The fourth line above is true because the dual of $af(w-c) + b\cdot(w-c)$ is $af^*((x-b)/a) + x\cdot c$. The fifth is true because $L_t^*(0) = -L_t(w_t)$. The last line is true because $g^\circ(-\eta_t m_t'(w_t))^2 = \eta_t^2 g^\circ(-m_t'(w_t))^2$.

The desired result now follows by applying Theorem 3.1 to the loss functions $l_0, m_1, m_2, \ldots$, taking $\tilde{l}_t = m_t$ and $c_t = \frac{1}{1-\eta_t}$. □

The way it is stated, this theorem bounds the loss in terms of the functions m_t; it is just as easy to give a bound in terms of l_t by substituting $m_t(w_t) = \mathbb{D}_{l_t}(w_t|a_t)$ and $m_t(u) \leq \mathbb{D}_{l_t}(u|a_t)$.

3.8 GGD examples

Perhaps the simplest use of GGD is to approximate the mean of a population of vectors by looking at a sample $z_1, z_2, \ldots$. This application of GGD corresponds to the prior loss $l_0(w) = k||w||^2$ and the one-step losses $l_t(w) = ||w - z_t||^2$. With these loss functions, GGD will predict $w_{t+1} = w_t + \frac{1}{k}(z_t - w_t)$. We saw above that we can take g to be Euclidean distance and $\eta = \frac{1}{k}$; so Theorem 3.2 tells us that our loss is bounded by

$$\sum_{t=1}^{T} ||w_t - z_t||^2 \leq \frac{1}{1-\frac{1}{k}} \sum_{t=1}^{T} ||z_t - \bar{z}||^2 + \frac{k}{1-\frac{1}{k}} ||\bar{z}||^2$$

where $\bar{z} = \frac{1}{T+k}\sum_{t=1}^{T} z_t$ is the optimal constant prediction.

The first term on the right-hand side of the above inequality depends on the training examples z_t only through their variance; the second depends on the examples only through their mean. So, the inequality tells us that even if the training examples are chosen by an adversary, as long as they have bounded mean and variance, we can still achieve bounded average regret per trial. More specifically, suppose that as $T \to \infty$ the mean of $z_1 \ldots z_T$ approaches μ and the covariance approaches $\sigma^2 I$. Then for large enough T the second term becomes negligible, and our average loss per trial will approach $\frac{\sigma^2}{1-\eta}$. So, our average regret per trial will approach $\frac{\sigma^2\eta}{1-\eta}$.

By way of comparison, we can compute the asymptotic average case regret per trial for this variant of GGD: suppose that the training examples z_t are independent indentically distributed random variables that follow a normal distribution with mean μ and covariance $\sigma^2 I$. Then the optimal prediction will approach μ for sufficiently large T, and its expected loss on each trial will approach σ^2. On the other hand, by solving the recurrences $Ew_{t+1} = (1-\eta)Ew_t + \eta Ez_t$ and $\operatorname{Var} w_{t+1} = (1-\eta)^2 \operatorname{Var} w_t + \eta^2 \operatorname{Var} z_t$, we can see that $Ew_t \to \mu$ and $\operatorname{Var} w_t \to \frac{\eta}{2-\eta}\sigma^2 I$. So, the expected loss per trial of the GGD algorithm approaches

$$E\|z_t - \mu\|_2^2 + E\|\mu - w_t\|_2^2 \to \sigma^2(1 + \frac{\eta}{2-\eta}) = \frac{\sigma^2}{1-\frac{\eta}{2}}$$

and the average regret per trial approaches $\frac{\sigma^2\eta}{2-\eta}$. That means that as $\eta \to 0$ there is a difference of approximately a factor of two between the worst-case and average-case regret for this algorithm. This gap appears to be necessary: at least for small learning rates, the sequence $z_1, z_2, \ldots = 1, -1, 1, -1, \ldots$ forces nearly as much regret as our bound.

For another example, take l_0 to be a multiple of the entropy function on the unit simplex. That is, suppose $l_0(w) = kD_H(w|0)$, with H defined in Equation 3.4. The resulting update is

$$w_{t+1,i} = \frac{w_{t,i}\exp(-x_{t,i}/k)}{\sum_{i=1}^{N} w_{t,i}\exp(-x_{t,i}/k)}$$

where $x_t = l_t'(w_t)$. This is the Exponentiated Gradient algorithm of [KW97]. (If the loss functions l_t for $t \geq 1$ are linear, it is also the same as the WMC algorithm.)

If now $l_t(w) = \frac{1}{2}(w - z_t)^2$ for $t \geq 1$, we saw above that we can take $\eta = \frac{1}{4k}$. So Theorem 3.2 tells us that our loss is bounded by

$$\sum_{t=1}^{T} ||w_t - z_t||^2 \leq \frac{1}{1-\frac{1}{4k}} \sum_{t=1}^{T} ||u - z_t||^2 + \frac{k}{1-\frac{1}{4k}} \mathbb{D}_H(u|0)$$

for any u. This bound is not the same as any bound in [KW97] or [KW96], since those papers consider only regression problems; so, we defer a comparison until Section 3.11 below.

3.9 Inference in exponential families

The MAP algorithm requires solving a minimization problem to find each prediction w_t. If the loss functions are arbitrary, the minimization problem may be difficult. Suppose, though, that l_t has the same functional form for each t—say, $l_t(w) = \mathbb{D}_l(w|a_t)$ for some fixed strictly convex function l. (By convention we take l so that $a_0 = 0$.) Then, as we will show shortly, we will always be able to put the optimization problem into a simple form.

One situation where this kind of prediction problem might arise is when the vectors a_t are samples from some target distribution. Our goal in this case is to predict w_t so that $l'(w_t)$ is as close as possible to the center of the distribution, where centrality is defined by the divergence $\mathbb{D}_l$. As we will see in Section 3.9.2, this definition of centrality is a good one if we are trying to infer the natural parameter in an exponential family of distributions (hence the title of this section). Unlike the standard statistical approach, though, we are making no distributional assumptions about the vectors a_t: they need not be identically distributed, independent, or even random.

In more detail, our optimization problem at step t is to find

$$\arg\min_w \sum_{i=0}^{t-1} l_i(w)$$

Define $L_t = \sum_{i=0}^{t-1} l_i$, so that our problem is to minimize L_t. Then the prediction of the MAP algorithm will be

$$\begin{aligned}
\arg\min_w L_t(w) &= \arg\min_w \sum_{i=0}^{t-1} [l(w) + l^*(a_i) - w \cdot a_i] \\
&= \arg\min_w \left[tl(w) - w \cdot \sum_{i=0}^{t-1} a_i \right] \\
&= \arg\min_w \left[l(w) - w \cdot \frac{1}{t} \sum_{i=0}^{t-1} a_i \right] \\
w &\in \partial l^*(\bar{a}_t)
\end{aligned}$$

where we have defined $\bar{a}_t$ to be the mean of $a_0 \dots a_{t-1}$. In other words, the MAP algorithm has a simple implementation: to make our prediction, we compute the average of all the samples a_t seen so far, then apply $(l^*)'$ to this average.

The implementation is almost the same if we take $l_0 = n_0 \mathbb{D}_l(w|a_0)$ for some multiplier n_0. In that case, the prediction is a weighted average of $a_0 \dots a_{t-1}$ in which a_0 gets n_0 times as much weight as any of the other a_is.

3.9.1 Regret bounds

In our current inference problem, L_t and l_t each differ from a multiple of l by a linear function. So, in order to apply Theorem 3.2, we must show that l is neither too convex nor too shallow. In other words, we must find a gauge g and constant k so that

$$k\mathbb{D}_l(v|a) \geq \frac{1}{2}(g(v-w))^2$$

for all v and w and $a \in k\partial l(w)$ and

$$\mathbb{D}_{l^*}(a_t|w) \geq \frac{1}{2}(g^\circ(a_t-a))^2$$

for all w and $a \in \partial l(w)$.

Under these assumptions, we can apply Theorem 3.2 with $l_0 = n_0\mathbb{D}_l(w|0)$, $l_t(w) = m_t(w) = \mathbb{D}_l(w|a_t)$ for $t \geq 1$, and $\eta_t = \frac{k}{n_0+t-1}$. (To make sure that $\eta_t < 1$ we must take $n_0 > k$.) The result is that

$$\sum_{t=1}^{T} \mathbb{D}_l(w_t|a_t) \leq \sum_{t=1}^{T} \frac{1}{1-\eta_t}\mathbb{D}_l(u|a_t) + \frac{n_0}{1-\eta_1}\mathbb{D}_l(u|0)$$

If $l_t(u)$ is bounded for t larger than some t_0, then the first term on the right hand side is $O(t + \ln t)$ as $t \to \infty$. This is the same asymptotic behavior as the average-case regret, although the constant in front of $\ln t$ will usually be smaller for average- than for worst-case bounds.

The constant k will be equal to 1 only if l is quadratic. However, if the predictions w_t remain in some region W for sufficiently large t, for those t we can take g and k as bounds on the convexity of l just within W instead of globally. This trick may result in better asymptotic bounds in some cases. Even with this trick the bounds may not be very tight: for example, it does not appear to be possible to prove bounds of the form obtained in [Fre96] using this strategy.

3.9.2 A Bayesian interpretation

We have just proved worst-case regret bounds for a special case of the MAP algorithm. Interestingly, we can also justify the same algorithm with an average-case argument. (For background see [BN78].) Suppose, just as before, that our loss on step t is $l_t(w) = \mathbb{D}_l(w|a_t)$. Suppose now, though, that each a_t is an independent sample from some known distribution. To ensure that the loss is finite, we will require a_t to be in $\operatorname{dom} l^*$ w.p.1.

In particular, suppose that the distribution of a_t has the form

$$\mu(a|\theta) = \exp(\theta \cdot a - \kappa(\theta) - \phi(a))$$

for some parameter vector θ and fixed functions κ and ϕ. (Such a set of distributions is called an exponential family, and θ is called its natural parameter.) Suppose also that our prior distribution for θ has the form

$$\nu(\theta|\lambda_0, n_0) = \exp(\lambda_0 \cdot \theta - n_0\kappa(\theta) - \chi(\lambda_0, n_0))$$

for parameters λ_0 and n_0, where the function χ is determined by the requirement that the density must integrate to 1. (This distribution is called the conjugate prior for μ, and it is also an exponential family.) We will see below that choosing $\kappa = l$ has an intuitive interpretation, but other choices of κ may also be reasonable.

Then the log posterior likelihood after seeing t samples will be

$$\theta \cdot \lambda - n\kappa(\theta)$$

where $\lambda = \lambda_0 + \sum_{i=1}^{t} a_i$ and $n = n_0 + t$. We can find the posterior distribution for θ by normalizing the posterior likelihood so it integrates to 1. In fact, by the definition of χ, the normalization factor is $\exp(-\chi(\lambda, n))$. So, the posterior for θ is $\nu(\theta|\lambda, n)$.

Notice that the posterior distribution of θ depends on the observed samples z_t only through $\sum_i a_i$. This sum is called a sufficient statistic for inference about θ, since once we know it we need no other information about the z_ts to compute the posterior distribution for θ.

Now that we have the posterior distribution for θ, we can compute the best prediction w. First suppose that we knew θ exactly. Then the expected loss on each step would be

$$E_\theta(l(w) + l^*(a) - w \cdot a)$$

where we have written E_θ as shorthand for $E(\cdot|a \sim \mu(a|\theta))$. Since we don't know θ exactly, we must take the expectation of the above expression under our posterior distribution for θ. That yields an expected loss of

$$l(w) + E_{\lambda,n}(E_\theta(l^*(a))) - w \cdot E_{\lambda,n}(E_\theta(a))$$

Since l is convex, we can find the w which minimizes expected loss by differentiating and setting to zero:

$$0 \in \partial l(w) - E_{\lambda,n}(E_\theta(a))$$

So, we can pick any w in $\partial l^*(E_{\lambda,n}(E_\theta(a)))$.

Technically, we need to worry that there might be no w that achieves the minimum. In that case the above equation would have no solution. But our reasoning below will provide conditions which guarantee that the expected value is always in $\operatorname{int}\operatorname{dom} l^*$. So, under those conditions a solution must exist.

Under some regularity conditions on μ, we can compute the expected value of a. First, we can prove by differentiating the identity $\int \mu(a|\theta) da = 1$ that

$$E_\theta(a) = \kappa'(\theta)$$

(see for example equation 2.2 (i) of [DY79]). For this reason, $\kappa'(\theta)$ is called the expectation parameter of the distribution μ. Next, by applying Theorem 2 of [DY79], we find that

$$E_{\lambda,n}(\kappa'(\theta)) = \frac{\lambda}{n}$$

Link name	Distribution	Conjugate prior
Identity	Normal	Normal
Logistic	Beta	Binomial
Inverse logistic	Binomial	Beta
Exponential	Poisson	Gamma
Logarithmic	Gamma	Poisson
Normalized exponential	Dirichlet	Multinomial

Figure 3.5: Links and their associated distributions.

So, as long as λ/n is in $\operatorname{int}\operatorname{dom} l^*$, there will be a legal prediction w. But λ/n will be in $\operatorname{int}\operatorname{dom} l^*$ as long as λ_0/n_0 is, since it's the average of a bunch of quantities in $\operatorname{dom} l^*$ at least one of which is in $\operatorname{int}\operatorname{dom} l^*$.

But now we have arrived back at our original algorithm: to find the prediction w_t, just average together $a_0 \ldots a_t$ and then apply $(l^*)'$. Interestingly, this conclusion doesn't depend on which exponential family we choose as the distribution for a_t. Instead, any exponential family which is contained in $\operatorname{dom} l^*$ results in the same optimal prediction. However, if we choose the exponential family so that $\kappa = l$, then we can interpret w as the inferred value of the natural parameter.

The mapping $(l^*)'$ which takes us from the observed average to the natural parameter is called a link function, just as it was for generalized gradient descent. Figure 3.4 above shows some useful link functions. Figure 3.5 shows which link functions correspond to which exponential families if we choose $\kappa = l$.

3.10 Regression problems

A common type of prediction problem is generalized linear regression [MN83, LW92, KW97], which includes linear regression, logistic regression, other generalized linear models, perceptron learning, and many other problems. In generalized linear regression, on each time step t we must predict a vector of regression coefficients w_t. We are then given an input vector x_t, from which we form a prediction $\hat{y}_t = f(x_t \cdot w_t)$. The monotone function f is called the prediction link function, since it provides a link between the coefficients w_t and the prediction $\hat{y}_t$. Finally, we find out the desired output y_t and receive a loss $l_t(w) = l(\hat{y}_t, y_t)$. Regression problems are a special case of our general prediction problem, since they differ only in that we have specified a particular form for the loss function l_t: for example, the loss functions for linear regression are of the form $(y_t - w \cdot x_t)^2$.

We should not confuse the prediction link function, which is a mapping from $\mathbb{R}$ to $\mathbb{R}$ that connects $w \cdot x_t$ with the prediction $\hat{y}$, with the link function described earlier, which is a function from $\mathcal{W}$ to $\mathcal{W}$ that connects the natural parameters with the expectation parameters. In designing an algorithm, we can choose the two kinds of link functions separately. When there is a danger of confusion,

Name	Link Function f	Corresponding F
Step	$\begin{cases} -1 & p < 0 \\ 1 & p \geq 0 \end{cases}$	$\lvert p \rvert$
ϵ-insensitive	$\begin{cases} -1 & p < -\epsilon \\ 0 & -\epsilon \leq p \leq \epsilon \\ 1 & p > \epsilon \end{cases}$	$\max(\lvert p \rvert - \epsilon, 0)$
Huber	$\begin{cases} -1 & p < -\epsilon \\ p/\epsilon & -\epsilon \leq p \leq \epsilon \\ 1 & p > \epsilon \end{cases}$	$\begin{cases} -p - \epsilon/2 & p < -\epsilon \\ p^2/2\epsilon & -\epsilon \leq p \leq \epsilon \\ p - \epsilon/2 & p > \epsilon \end{cases}$

Figure 3.6: Some examples of prediction link functions.

we will call the latter the parameter link function. All of the one-dimensional parameter link functions in Figure 3.4 are also possible choices for the prediction link function; Figure 3.6 shows some additional possible choices.

3.10.1 Matching loss functions

In order to apply our theory, we need the one-step losses $l_t(w)$ to be convex. This is a condition on the relationship between the prediction link function f and the loss function $l(\hat{y}, y)$. It turns out that, given a monotone link function, we can always define a matching loss function so that $l_t(w)$ is convex. If f is invertible, we follow [AHW96] and define its matching loss function to be

$$l(\hat{y}, y) = D_{F^*}(y|\hat{y})$$

where F is any convex function with $f = F'$.

If f is not invertible (that is, if F has a linear segment, so that F^* has a corner) then the above definition no longer works. Intuitively, the problem is that our predictions get "stuck" as they cross the corner in F^*: there is a whole range of p with the same $f(p)$ and therefore the same loss, producing an extraneous flat spot in l_t.

We can fix the problem by allowing $l_t(w)$ to depend on $p_t = x_t \cdot w$ directly, rather than just on $f(p_t)$. More specifically, we generalize the definition of [AHW96] and set

$$m(p, y) = \mathbb{D}_{F^*}(y|p) = \mathbb{D}_F(p|y) = F(p) + F^*(y) - y \cdot p$$

With this definition, it is easy to see that $m(p, y)$ is convex as a function of p, so $l_t(w) = m(x_t \cdot w, y_t)$ is convex in w. Intuitively, what we have done is allow ourselves to specify not just which y will give us zero loss, but also what the derivative of F^* is at that point. When F^* is smooth, there is only one possible choice of derivative for each prediction, so we have not changed anything; but when our prediction is at a corner of F^* we can choose from a range of possible derivatives.

We will use the derivative of the loss function below. It turns out that the prediction error is a derivative of m with respect to p:

$$f(p) - y \in \partial F(p) - y = \partial_p m(p, y)$$

So, a derivative of $l_t(w)$ is $(f(x_t \cdot w) - y_t)x_t$.

3.10.2 Regret bounds

In order to bound the regret of the MAP algorithm for regression problems, we need to find a gauge g so that $l_t(w) \geq \frac{1}{2}(g^\circ(-l_t'(w)))^2$. We have already done so for the special case of the identity link with squared loss: in section 3.7.2, we showed that the allowable choices for g are the symmetric gauges such that $g^\circ(x_t) \leq 1$ for all t. (Symmetric gauges are also called seminorms.)

The situation is similar for general link functions and their matching loss functions. In this case, though, we must make one additional assumption: we must bound how quickly the prediction $\hat{y}_t$ changes when we change the raw prediction p_t.

So, we will assume that $\mathbb{D}_F(p|y) \geq \frac{\lambda^2}{2}(y - f(p))^2$ for some $\lambda > 0$. (This is essentially a Lipschitz condition on f.) With this assumption, we can write

$$\begin{aligned} l_t(w) &= \mathbb{D}_F(x_t \cdot w|y_t) \\ &\geq \frac{\lambda^2}{2}(y_t - f(x_t \cdot w))^2 \end{aligned}$$

But we saw above that $l_t'(w) = (f(x_t \cdot w) - y_t)x_t$. So, if g is a symmetric gauge such that $\lambda g^\circ(x_t) \leq 1$, then

$$\begin{aligned} (y_t - f(x_t \cdot w))^2 &\geq \frac{1}{\lambda^2} g^\circ(l_t'(w))^2 \\ l_t(w) &\geq \frac{1}{2} g^\circ(l_t'(w))^2 \end{aligned}$$

But now we have proven

Theorem 3.3 *Let F be a closed convex function with $\mathbb{D}_F(p|y) \geq \frac{\lambda^2}{2}(y - f(p))^2$. Suppose that the functions $l_1, l_2, \ldots$ are of the form $l_t(w) = \mathbb{D}_F(y_t|w \cdot x_t)$ for given vectors x_t and scalars y_t. Pick a prior loss l_0 and functions $m_1, m_2, \ldots$, and suppose that $l_0, m_1, m_2, \ldots$ satisfy the conditions of Lemma 3.1, so that the MAP algorithm applied to these loss functions always produces a prediction w_t at each trial. Suppose that for all t, $\partial m_t(w_t)$ is nonempty, $m_t(w_t) = l_t(w_t)$, and $m_t(w) \leq l_t(w)$ for all w. Write $L_t = l_0 + \sum_{i=1}^{t-1} m_t$. Let the symmetric gauge g be so that $\lambda g^\circ(x_t) \leq 1$ for all t. Finally, let the constants $1 \geq \eta_1 \geq \eta_2 \geq \ldots > 0$ be such that*

$$\eta_t \mathbb{D}_{L_t}(v|a) \geq \frac{1}{2}(g(v - w))^2$$

for all v and w and $a \in \partial L_t(w)$. Then

$$\sum_{t=1}^{T} l_t(w_t) \leq \sum_{t=1}^{T} \frac{1}{1 - \eta_t} l_t(u) + \frac{1}{1 - \eta_1} \mathbb{D}_{l_0}(u|0)$$

for all u.

PROOF: Apply Theorem 3.2 to the functions $l_0, l_1, \ldots, m_1, m_2, \ldots$, with $a_t = 0$, using the gauge g and the learning rates η_t. $\square$

While the size of the input vectors x_t doesn't appear explicitly in this bound, it affects the choice of g and therefore the allowed values of η_t. For example, depending on the size of the input vectors, we might need to set $g(w)$ to either $||w||_1$ or $10||w||_1$. At the cost of introducing an extra parameter, we could have written the theorem to allow us to set $g(w) = ||w||_1$ no matter the scale of the input vectors. For examples of the application of this theorem, see Section 3.11.

3.10.3 Multidimensional outputs

So far in our regression problems we have assumed that the target y_t is one-dimensional. Our proofs work equally well, though, if y_t is selected from an arbitrary vector space $\mathcal{Y}$. In that case, the parameter matrix w_t will be a linear mapping that takes x_t to $p_t \in \mathcal{Y}$. The prediction link function f will be the derivative of some convex function F on $\mathcal{Y}$, and the matching loss will be $\mathbb{D}_F(p_t|y_t)$, so the derivative of $l_t(w)$ will be $(f(wx_t) - y_t)x_t^{\mathrm{T}}$.

The only part of the proof that requires some modification is the definition of the gauge g. Since w is a matrix and x_t and y_t are vectors of possibly different lengths, we need different gauges to measure the size of each one. (Previously we had used g for w, g° for x_t, and $|\cdot|$ for y_t.) So, we will assume that we have symmetric gauges r and s so that $r^\circ(x_t) \leq 1$ and $\mathbb{D}_F(p|y) \geq \frac{1}{2}s(y - f(p))^2$. Then we will define g by the relation

$$g(w) = \sup_{\{u|r(u)\leq 1\}} s(wu)$$

This g is called the matrix gauge for r and s. Since r and s are symmetric, so is g. Also, if x and y are vectors with $r^\circ(x) = 1$ and $s(y) = 1$, then $g(yx^{\mathrm{T}}) = 1$, since $s(yx^{\mathrm{T}}u) = s(y)x^{\mathrm{T}}u \leq s(y)r^\circ(x)r(u)$, with equality for an appropriately chosen u.

With this choice of g, we have

$$\begin{aligned}
\frac{1}{2}g(l_t'(w))^2 &= \frac{1}{2}g((f(wx_t) - y_t)x_t^{\mathrm{T}})^2 \\
&= r^\circ(x_t)^2\frac{1}{2}s(f(wx_t) - y_t)^2 \\
&\leq \mathbb{D}_F(wx_t|y_t) \\
&= l_t(w)
\end{aligned}$$

so Theorem 3.3 applies with $\lambda = 1$. (To achieve the effect of varying λ we can simply rescale the gauge s.)

For example, if f is the identity prediction link (so that $F(y) = \frac{1}{2}y^{\mathrm{T}}y$ and we can take s to be Euclidean distance) and $||x_t||_2 \leq 1$ (so that we can take r to be Euclidean distance also), then $g(w)$ will be the matrix 2-norm $||w||_2$. If we now take $l_0(w) = \frac{k}{2}\sum_{i,j} w_{ij}^2$, then $\frac{1}{k}D_{l_0}(v|w) \geq \frac{1}{2}g(v-w)^2$, so we can take $\eta_1 = \frac{1}{k}$.

Often we will take each coordinate of f to be one of the one-dimensional link functions described above. This kind of link function decomposes the multiple-output prediction problem into several single-output problems which share a parameter vector. On the other hand, sometimes we may know about dependencies among the components of the output vector. In this case we can take advantage of our knowledge by picking a prediction link function that encodes these dependencies. For example, if we have reason to believe that the output vector has covariance matrix Σ, we can select the link $\hat{y} = \Sigma p$ with its matching loss $\frac{1}{2}\hat{y}^{\mathrm{T}}\Sigma^{-1}\hat{y} - \hat{y}^{\mathrm{T}}p + \frac{1}{2}p^{\mathrm{T}}\Sigma p$.

3.11 Linear regression algorithms

In this section we will analyze several gradient-descent-like algorithms for linear regression: standard gradient descent and two exponentiated gradient algorithms from [KW97] called EG and $\mathrm{EG}^{\pm}$. These algorithms are all generalized linear regression algorithms, and therefore MAP algorithms.

In linear regression problems, the loss on trial $t \geq 1$ is $l_t(w) = \mathbb{D}_{l_t}(w|0) = \frac{1}{2}(y_t - x_t \cdot w)^2$. This is the loss function for a generalized linear regression model using the identity prediction link with its matching loss function, the squared error. The algorithms differ only in their choice of prior loss l_0.

We will bound the regret of each algorithm by applying Theorem 3.3. Because l_t for $t \geq 1$ always has the form given above, we can take $\lambda = 1$ in Theorem 3.3; so the main part of the analysis of each algorithm will be to find appropriate seminorms g and g° with which to measure the parameter vectors w_t and the input vectors x_t.

First consider the gradient descent algorithm for linear regression, defined by the update

$$w_{t+1} = w_t + \eta(y_t - w_t \cdot x_t)x_t$$

Gradient descent is a GGD algorithm, given by the choice $l_0(w) = \frac{1}{2\eta}||w||_2^2$ (or $l_0 = \frac{1}{2\eta}||w - w_1||_2^2$ if we want a starting weight vector $w_1 \neq 0$). We showed above that if $||x_t||_2 \leq X$ for all t then we can take $g^\circ(x) = \frac{1}{X}||x||_2$ and $\eta_t = X^2\eta$ in Theorem 3.3. The result is that

$$\sum_{t=1}^{T} l_t(w_t) \leq \frac{1}{1 - X^2\eta}\sum_{t=1}^{T} l_t(u) + \frac{1}{2\eta(1 - X^2\eta)}||u||_2^2$$

Next consider the exponentiated gradient algorithm. EG is a GGD algorithm given by the choice $l_0(w) = \frac{1}{\eta}H(w)$, so its update is

$$v_{t+1,i} = w_{t,i}\exp(\eta(y - w_t \cdot x_t)x_{t,i})$$

$$w_{t+1,i} = \frac{v_{t+1,i}}{\sum_j v_{t+1,j}}$$

To analyze EG, we will set X to be the maximum span of any of the input vectors, that is, $||x_t||_{\text{sp}} \leq X$ where

$$||x||_{\text{sp}} = \max_i x_i - \min_i x_i$$

It is easy to check that $||\cdot||_{\text{sp}}$ is a seminorm. We can bound the polar of the span seminorm by splitting its argument vector into components parallel and perpendicular to $e = (1, 1, \ldots, 1)^{\text{T}}$. We have $||e||_{\text{sp}} = 0$, so $||e||_{sp}^{\circ} = \infty$. On the other hand, if x has no component along e, then $||x||_{\text{sp}} \geq ||x||_\infty$, so $||x||_{\text{sp}}^{\circ} \leq ||x||_1 \leq ||x||_2$. That means that, for any v and w and $a \in \partial H(w)$,

$$\mathbb{D}_H(v|a) \geq 2(||v - w||_{\text{sp}}^{\circ})^2$$

To see why, remember that by assumption $\partial H(w)$ is nonempty, so w must be in the unit simplex. So, depending on whether v is in the plane containing the unit simplex, either $v - w$ has a nonzero component along e, in which case $\mathbb{D}_H(v|a)$ is infinite, or $v - w$ is perpendicular to e, in which case $\mathbb{D}_H \geq 2||v - w||_2^2$. In either case the result follows. So, we can take $g^{\circ}(x) = \frac{1}{X}||x||_{\text{sp}}$ and $\eta_t = \frac{1}{4}X^2\eta$ in Theorem 3.3 and conclude that

$$\sum_{t=1}^{T} l_t(w_t) \leq \frac{1}{1 - \frac{1}{4}X^2\eta} \sum_{t=1}^{T} l_t(u) + \frac{1}{\eta(1 - \frac{1}{4}X^2\eta)} H(u)$$

The above results can be compared to to Lemmas 5.2 (for GD) and 5.9 (for EG) in [KW97]. Unfortunately, our bounds here are slightly weaker than the ones in [KW97]. We do not believe that this is due to a weakness in our framework; instead we believe that with some additional work our theorems could be sharpened so that they are a strict generalization of the known results for linear regression with GD and EG.

After deriving the results mentioned above, the authors of [KW97] perform an additional step: they adjust the learning rate η so that the two terms in the regret bound have comparable coefficients. We have not taken this step.

Finally consider the EG$^{\pm}$ algorithm. Just as in [KW97], we could prove bounds on EG$^{\pm}$ by reducing it to EG. Instead, we will sketch how to find the prior l_0 that yields the EG$^{\pm}$ algorithm. Finding this prior is important both because it increases our understanding of EG$^{\pm}$ and because it is a good first step towards a direct proof of the regret bound for EG$^{\pm}$.

The EG$^{\pm}$ algorithm can be defined by its parameter link function, which is (up to scaling) given by the mapping $w = f(x)$ defined as

$$w_i = \frac{\exp(x_i) - \exp(-x_i)}{\sum_j (\exp(x_j) + \exp(-x_j))}$$

The prior loss function l_0 for EG$^{\pm}$, and its convex dual l_0^*, are determined up to scaling by this link function. We can find $l_0^*(x)$ by integrating f along any path from the origin to x.

To perform the integral, we will choose a path with n axis-parallel segments: one which increases the first coordinate of x from 0 to its final value x_1, then another which increases the second coordinate from 0 to its final value x_2, and so forth. The integral along the ith segment (which varies the ith coordinate of x) is

$$\int_0^{x_i} \frac{\exp(t) - \exp(-t)}{\exp(t) + \exp(-t) + k_i} dt$$

where the constant

$$k_i = \sum_{j=1}^{i-1} (\exp(x_j) + \exp(-x_j))$$

is determined by the (constant) values of the other $n-1$ coordinates of x along the ith segment. The result of this integral is

$$-x_i - \ln(2 + k_i) + \ln(1 + \exp(2x_i) + k_i \exp(x_i))$$

Summing this expression over all n path segments gives

$$l_0^*(x) = \sum_{i=1}^{n} (-x_i - \ln(2 + k_i) + \ln(1 + \exp(2x_i) + k_i \exp(x_i)))$$

For example, if $n = 2$,

$$\begin{aligned} l_0^*(x) \quad = \quad & -x_1 - x_2 - \ln 4 + \ln(1 + \exp(2x_2) + (\exp(x_1) + \exp(-x_1)) \exp(x_2)) \\ & - \ln(2 + \exp(x_1) + \exp(-x_1)) + \ln(1 + 2\exp(x_1) + \exp(2x_1)) \end{aligned}$$

A plot of this function is in the left panel of Figure 3.7; it looks like a rounded-off version of the L_∞ norm. The right panel of Figure 3.7 shows a plot of the three-dimensional version of l_0^*, made by holding one argument constant at 7 while varying the other two in $[-10, 10]$. In other words, we have plotted l_0 on a two-dimensional slice of $\mathbb{R}^3$. This plot looks like a rounded-off version of the same slice of the L_∞ norm on $\mathbb{R}^3$. The characterization of l_0^* as a rounded-off version of the L_∞ norm makes sense, since EG$^\pm$ restricts w_t to have bounded L_1 norm and the dual of $\delta(x \mid ||x||_1 \leq 1)$ is the L_∞ norm.

3.12 Discussion

We have presented a unified framework for deriving worst-case regret bounds for a wide class of learning algorithms. These algorithms include weighted majority; gradient descent and generalizations of gradient descent such as exponentiated gradient; linear and logistic regression; and inference of the natural parameter in an exponential family. Because we have wherever possible avoided assumptions such as differentiability of the loss functions, our framework also includes a wide variety of new algorithms which we have not fully explored.

Our unified treatment sheds light on the relationships among these methods, and it provides a recipe for designing and studying new learning algorithms. For

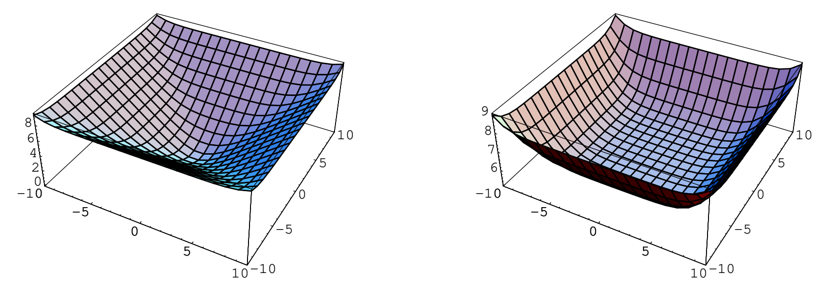


Figure 3.7: The dual of the potential function for $EG^{\pm}$.

example, we showed that both the gradient descent and exponentiated gradient algorithms for linear regression are MAP algorithms. By casting them in this common framework, we revealed that the only difference between these two algorithms is their choice of prior loss function. In addition to allowing a common proof of the regret bounds for these algorithms, this analysis suggests that we can design new linear regression algorithms simply by picking new priors. These priors can express known bounds on the parameter vector (for example, the prior $kw^2 + \delta(w|C)$ yields the gradient projection algorithm with domain C) or preferences for particular kinds of parameter vectors (for example, the prior of the $EG^{\pm}$ algorithm prefers vectors with low L_1 norm).

Our results also suggest new applications for old algorithms. By avoiding assumptions such as independence of training examples, we have justified the use of these algorithms in situations where they might not have been considered before.

Chapter 4

CONVEX ANALYSIS AND MDPS

In this chapter we will apply the ideas of convex analysis and statistical inference to the problem of approximating the value function of a Markov decision process. In Sections 4.1 through 4.4, we will show how to represent an MDP as a convex program. This transformation will allow us to apply the well-known theory of convex programming to the problem of finding its value function. In Section 4.5 we will show how to represent an MDP as either a maximum likelihood or a maximum entropy problem. This transformation will allow us to apply the well-known theory of statistical inference to the problem of finding its value function. In Section 4.6, we will describe several ways we have tried to introduce approximations of the value function into these two representations of an MDP. In Section 4.7 we will describe an implementation of one of these algorithms. Finally, in Section 4.8, we will describe some experiments we have done with this implementation. While the experiments show that this particular algorithm does not improve on the best existing ones, we hope that the ideas of this chapter can be incorporated into other algorithms.

4.1 The Bellman equations

We saw in Chapter 1 that the value function for an MDP is the unique solution to the Bellman equations

$$v(x) = \min_{a \in A} \mathbb{E}(c(x,a) + \gamma v(\delta(x,a)))$$

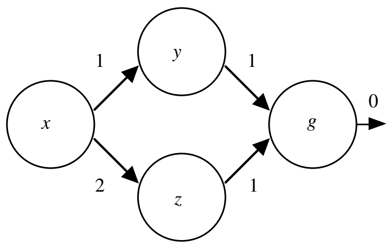
(base cases such as $v(\odot) = 0$ may be necessary if $\gamma = 1$). As pointed out in for example [Ros83, p40] or [Ber76, p248], we can rewrite the Bellman equations as a linear program by noticing:

- If there's a deterministic action a that takes the agent from state x to state y with cost $c(x,a)$, then $v(x) \leq \gamma v(y) + c(x,a)$.
- Similarly, if there's a stochastic action a that takes the agent from state x to a probability distribution p over the state space, then $v(x) \leq \gamma p \cdot v + c(x,a)$. The notation $p \cdot v$ means $\sum_y v(y)p(y)$; in other words, $p \cdot v$ is the expectation of the value of the next state.
- The value function is the pointwise largest function v satisfying these constraints along with any base cases.

The resulting linear program is

$$\begin{array}{ll} \text{maximize} & s^{\mathrm{T}} v \\ \text{subject to} & Ev + c \geq 0 \end{array}$$

where E is the edge adjacency matrix for our MDP (defined in Chapter 1), c is the cost vector, and s is any vector with all components positive. For this section we will assume that s has all components equal to 1; in Section 4.2.3 we will attach a meaning to the choice of objective vector.



(a)

maximize $x + y + z + g$ subject to

$$
\begin{array}{rrrrrcl}
-x & +y & & & +1 & \geq & 0 \\
-x & & +z & & +2 & \geq & 0 \\
& -y & & +g & +1 & \geq & 0 \\
& & -z & +g & +1 & \geq & 0 \\
& & & -g & & \geq & 0
\end{array}
$$

(b)

Figure 4.1: How to turn an MDP into an LP.

Figure 4.1 shows an example of translating a simple MDP to a linear program. (To avoid clutter we have adopted the shorthand of writing just x instead of $v(x)$ to mean the value of state x.) This MDP happens to be undiscounted and deterministic, but the translation works just as well for discounted or stochastic MDPs. There is one constraint in the program (that is, one row of E) for each edge or state-action pair in the MDP. There is one variable in the program (that is, one column of E) for each state in the MDP. For example, the row $-x + y + 1 \geq 0$ corresponds to the edge from state x to state y with cost 1. If there were a unit-cost action that moved the agent from state x to state y with probability .7, and from state x to state z with probability .3, the corresponding constraint would be $-x + .7y + .3z + 1 \geq 0$.

The optimal solution to this MDP is $(x, y, z, g) = (2, 1, 1, 0)$. In linear programming terminology, the elements of the vector $Ev + c = (0, 1, 0, 0, 0)^{\mathrm{T}}$ are called slacks; in dynamic programming terminology, they are called advantages or Bellman residuals. In either case, the edges in the optimal policy are the ones whose slack is 0. That means that an optimal policy for the MDP is the same as an optimal basis for the linear program. (This is a consequence of the property called complementary slackness.)

4.2 The dual of the Bellman equations

4.2.1 Linear programming duality

Every linear program can be paired with another linear program called its dual. The original (or primal) and dual programs are different views of the same problem: the optimal values of their objective functions are the same, and knowing a solution to one makes it much easier to find a solution to the other.

We can derive linear programming duality by appealing to duality between convex functions. Consider the linear program

$$\text{minimize } c^{\mathrm{T}} x \text{ subject to } Ax + b = 0, x \geq 0$$

We can eliminate the equality constraints by adding a vector y of Lagrange multipliers. So, solving the linear program is equivalent to finding

$$\min_x \max_y([c^{\mathrm{T}}x + y^{\mathrm{T}}(Ax + b)] + \delta(x|x \geq 0)) \tag{4.1}$$

The notation $\delta(x|x \geq 0)$ is defined in Chapter 3; it stands for the function which is zero if $x \geq 0$ and ∞ otherwise. The expression in square brackets is called the Lagrangian of the linear program. If the program has a finite solution, then we may interchange the order of minimization and maximization to get

$$\begin{aligned}
&\max_y \min_x([c^{\mathrm{T}}x + y^{\mathrm{T}}(Ax + b)] + \delta(x|x \geq 0)) \\
&= \; -\min_y \max_x[-c^{\mathrm{T}}x - y^{\mathrm{T}}Ax - y^{\mathrm{T}}b - \delta(x|x \geq 0)] \\
&= \; -\min_y \left[-y^{\mathrm{T}}b + \max_x[-x^{\mathrm{T}}(A^{\mathrm{T}}y + c) - \delta(x|x \geq 0)]\right] \\
&= \; -\min_y[-y^{\mathrm{T}}b + (\delta(x|x \geq 0))^*(-(A^{\mathrm{T}}y + c)) \\
&= \; \max_y[y^{\mathrm{T}}b - \delta(y|A^{\mathrm{T}}y + c \geq 0)]
\end{aligned}$$

In other words, we may find the optimal objective value for our original linear program by solving the new linear program

$$\text{maximize } b^{\mathrm{T}}y \text{ subject to } A^{\mathrm{T}}y + c \geq 0$$

We define this new linear program to be the dual of our original program. If we replace $A^{\mathrm{T}}y + c \geq 0$ by $A^{\mathrm{T}}y + c = z, z \geq 0$ and then apply the same sequence of transformations, it is easy to verify that the result is equivalent to the primal program.

4.2.2 LPs and convex duality

When thinking about duality between linear programs, it is often useful to remember the specialization of the theory of convex duality to indicator functions. As defined in Section 3.2, the indicator function for a convex set C is

$$\delta(x|C) = \begin{cases} 0 & x \in C \\ \infty & x \notin C \end{cases}$$

This function is zero inside of C and infinite outside of C, so if we want to constrain the variable x in a minimization problem to be in the set C we can add $\delta(x|C)$ to the function to be minimized.

The simplest convex sets C are the cones. A cone is the set of all positive linear combinations of a set of vectors g_i called its generators. If we write G for the matrix with columns g_i, then $C = \{G\lambda | \lambda \geq 0\}$. Some examples of cones are the origin (generated by the empty set of generators), any linear subspace, and the two cones shown in Figure 4.2. If the set of generators is finite, then C is a closed convex set.

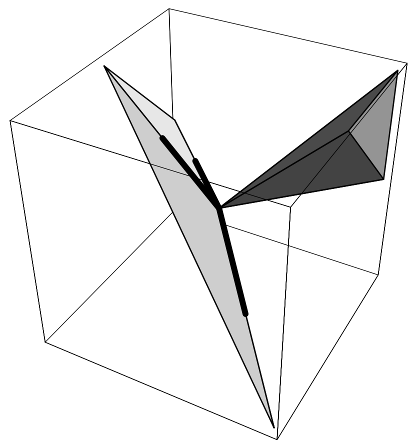


Figure 4.2: Two cones. Heavy lines show a set of generators for one of the cones.

The polar of a cone C, written C°, is the set of vectors which make either a right or an obtuse angle with every vector in C. That is,

$$C^\circ = \{x | (\forall y \in C)\ x \cdot y \leq 0\}$$

The polar is always a closed cone. For closed cones, the operation of taking the polar is its own inverse: the polar of the polar of a closed cone is the cone itself. The two cones in Figure 4.2 are polar to each other. As the figure shows, the extreme vectors of a cone are the face normals of its polar. Polarity between cones is an example of duality between convex functions: if C is a cone, then the dual of the indicator function $\delta(x|C)$ is $\delta(x|C^\circ)$.

We can represent an arbitrary convex set in $\mathbb{R}^n$ as the intersection of a cone in $\mathbb{R}^{n+1}$ with a fixed plane. For example, Figure 4.3 shows the representation of a triangle in $\mathbb{R}^2$ as the intersection of a cone and a plane in $\mathbb{R}^3$. Usually we will use the same coordinate system for $\mathbb{R}^{n+1}$ as we did for $\mathbb{R}^n$, with the addition of one extra coordinate (call it t). We can then identify $\mathbb{R}^n$ with the plane $t = 1$ in $\mathbb{R}^{n+1}$, so that we can represent the convex set C by the cone $\{(tx, t) | t \geq 0, x \in C\}$. This cone is called the homogeneous representation of C. If C is an affine set, then we can use either the regular homogeneous representation or the set $\{(tx, t) | t \in \mathbb{R}, x \in C\}$ called the affine homogenous representation of C.

We can now see that the familiar notion of geometric duality is a consequence of polarity between convex cones. Two affine subspaces C and D are defined to be geometrically dual if $x \cdot y = 1$ for all $x \in C$ and $y \in D$, while two arbitrary convex sets are defined to be geometrically dual if $x \cdot y \leq 1$ for $x \in C$ and $y \in D$. For example, the line $a \cdot x = 1$ and the point a are dual affine subspaces in $\mathbb{R}^2$,

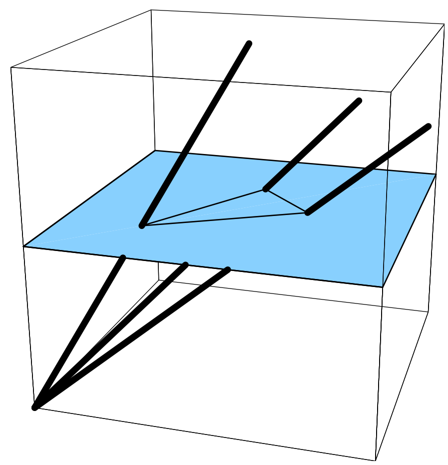


Figure 4.3: Homogeneous representation of a triangle.

while the unit cube and the unit octahedron are dual convex sets in $\mathbb{R}^3$. If C and D are geometrically dual convex sets, then the homogeneous representation of C is the polar of the homogeneous representation of D, reflected along the t axis. If C and D are geometrically dual affine sets, then the affine homogeneous representation of C is the polar of the affine homogeneous representation of D, reflected along the t axis.

We can take advantage of the connection between convex duality and cone polarity to analyze how operations on a cone change its polar. For example, intersection of two cones corresponds to addition of their indicator functions. The dual operation for addition is infimal convolution, defined as

$$(f \,\square\, g)(x) = \inf\{f(a) + g(b) | a + b = x\}$$

If f and g are indicator functions for the convex cones C and D, then $f(a)+g(b)$ will be zero if $a \in C$ and $b \in D$, and infinite otherwise. So, $(f \,\square\, g)(x)$ will be zero iff $x \in C + D$. That means that the polar of $C^\circ \cap D^\circ$ (corresponding to $f^* + g^*$) is $C + D$ (corresponding to $f \,\square\, g$); in other words, the operations of intersection and set sum are polar to each other.

For another example, if C and D are the homogeneous representations of convex sets, then we can write the intersection of $C^\circ + D^\circ$ with the plane $t = -1$ as the union of

$$\lambda(C^\circ \cap (t = -1)) + (1 - \lambda)(D^\circ \cap (t = -1))$$

for $\lambda \in [0, 1]$. In other words, the geometric dual of the intersection of two convex sets is the convex hull of the geometric duals of the sets.

The connection between a cone and its polar can help us understand the connection between a linear program and its dual. The relationship between a linear program and its dual is clearest for the degenerate linear program

$$\text{find } x \neq 0 \text{ such that } x \geq 0, Ax = 0 \tag{4.2}$$

This problem, called the homogeneous linear inequality problem, can be thought of as a linear program whose constant vectors are both zero. It is equivalent to asking whether there is an $x \neq 0$ for which the convex function

$$\delta(x|Q) + \delta(x|Ax = 0) \tag{4.3}$$

is zero, where Q is the nonnegative orthant. The dual problem to (4.2) is

$$\text{find } x \neq 0, y \text{ such that } x \leq 0, x = A^{\mathrm{T}}y$$

which corresponds to the question of whether there is an $x \neq 0$ for which the convex function

$$\delta(x| - Q) + \delta(x|x = A^{\mathrm{T}}y) \tag{4.4}$$

vanishes.

The expressions (4.3) and (4.4) are almost, but not quite, convex duals of each other. The dual of $\delta(x|Q)$ is $\delta(x| - Q)$, while the dual of $\delta(x|Ax = 0)$ is $\delta(x|x = A^{\mathrm{T}}y)$. But the dual operation to addition is infimal convolution, so the convex dual of (4.3) is

$$\delta(x| - Q) \,\square\, \delta(x|x = A^{\mathrm{T}}y)$$

which is the indicator function of the set $-Q + \{x|x = A^{\mathrm{T}}y\}$. In other words, there are four different convex sets associated with the system of inequalities (4.2): the intersection of the positive orthant with the linear constraint set, the sum of the positive orthant and the constraint set, the intersection of the dual of the positive orthant with the dual of the constraint set, and the sum of the dual of the positive orthant and the dual of the constraint set. Two of these four sets are the feasible regions for (4.2) and its dual, while the other two are the polars of the feasible regions.

For general linear programs the situation is similar: the difference is that instead of the indicator $\delta(x|Ax = 0)$ we have the function $\delta(x|Ax + b = 0) + c^{\mathrm{T}}x$, which is not an indicator function. (It is called a partial affine function, since its domain is an affine space and it is a linear function on its domain.) Still, we can construct four different convex functions by applying either addition or infimal convolution to either the indicators of Q and the partial affine function or their duals. Two of these functions represent the feasible region and objective function for the linear program and its dual.

4.2.3 The dual Bellman equations

The dual of the Bellman equation linear program is

$$\text{minimize} \qquad c^{\mathrm{T}}f$$

minimize $f_{xy} + 2f_{xz} + f_{yg} + f_{zg}$ subject to

$$\begin{array}{cccccccc} -f_{xy} & -f_{xz} & & & & +1 & = & 0 \\ f_{xy} & & -f_{yg} & & & +1 & = & 0 \\ & f_{xz} & & -f_{zg} & & +1 & = & 0 \\ & & f_{yg} & +f_{zg} & -f_g & +1 & = & 0 \end{array}$$

$f_{xy}, f_{xz}, f_{yg}, f_{zg}, f_g \geq 0$

Figure 4.4: The dual of the Bellman program.

$$\begin{array}{ll} \text{subject to} & E^{\mathrm{T}} f + s = 0 \\ & f \geq 0 \end{array}$$

This linear program has one equality constraint (that is, one row of E^{T}) for each state of our MDP, and one variable (that is, one column of E^{T}) for each edge or state-action pair of our MDP. The equality constraint for state y is

$$\sum_{x,a} p_{axy} f_{xa} + 1 = \sum_{a} f_{ya}$$

We can interpret f_{xa} as the expected number of times we visit the edge (x, a) if we follow one trajectory starting from each state. (If there is a discount factor, then f_{xa} is the expected discounted frequency.) We will call f_{xa} the flow along edge (x, a). Under this interpretation, the equality constraint for state y tells us that we must enter y exactly as often as we leave it. Since the objective function $f^{\mathrm{T}} c$ is equal to the expected cost of visiting the edge (x, a) a total of f_{xa} times, the dual Bellman program tells us to minimize the total expected cost of following one trajectory starting from each state.

Clearly it is not necessary to start exactly once in each state. If s is a vector of positive starting frequencies, so that we start $s_x > 0$ times in state x, then the equality constraint for state y becomes

$$\sum_{x,a} p_{axy} f_{xa} + s_y = \sum_{a} f_{ya}$$

The optimal vector of flows may be different for different choices of s, but the linear program will be feasible for any choice of $s > 0$. The fact that any positive vector of starting frequencies produces a feasible dual program is equivalent to the fact that any positive objective vector produces a bounded primal program.

Figure 4.4 shows the dual Bellman equation program for our example MDP from Figure 4.1. The optimal solution to this program is $(f_{xy}, f_{xz}, f_{yg}, f_{zg}, f_g) = (1, 0, 2, 1, 4)$. Just as with the primal program, if we know the optimal f we can find the best edge out of any state: any edge with positive flow will do.

4.3 Incremental computation

The previous sections describe how to convert a Markov decision process to a linear program. This transformation provides a simple algorithm for finding the value function of a known MDP: convert it to a linear program, then solve the linear program with, say, simplex or a logarithmic-barrier method. For some benchmarks of this algorithm versus value iteration, see [TZ93, TZ95, TZ97].

Often, though, we don't know the entire MDP in advance; or, even if we do know it, it is so large that we can't afford to examine every state even once. In either of these cases, we need an incremental version of the above algorithm. That is, we need to be able to convert a partly-known MDP into a linear program in such a way that when we solve the LP we end up with something close to the correct value function.

Incremental computation often goes hand in hand with approximation: if our MDP is so large that we need to look at it bit by bit, then we will often also need to use a compact representation for its value function. For this section, though, we will just worry about incremental computation, and leave approximation for Section 4.6. In other words, we will suppose that our MDP is small enough that we could solve it exactly if we knew it, but that we are finding out about it bit by bit.

There are at least two natural orders in which to reveal an MDP one piece at a time: edge by edge or state by state. Since every edge corresponds to a row of the adjacency matrix E, and since every state corresponds to a column of E, these two orders correspond to revealing E row by row or column by column.

We can represent either of these two orders, and many more, by writing E_t, c_t, and s_t for our best approximations to E, c, and s at time t. For example, if we are finding out about our MDP edge by edge, then $E_{t+1} - E_t$ will have nonzero entries in exactly one row.

With this notation, it is natural to suppose that the sequences f_t and v_t defined by the linear programs

$$\text{minimize } c_t^{\mathrm{T}} f_t \text{ subject to } E_t^{\mathrm{T}} f_t + s_t = 0, f_t \geq 0$$

$$\text{maximize } s_t^{\mathrm{T}} v_t \text{ subject to } E_t v_t + c_t \geq 0$$

might be good approximations to the optimal flows and values f and v respectively. Unfortunately, f_t and v_t do not necessarily converge to f and v even if $E_t \to E$, $c_t \to c$, and $s_t \to s$. For example, a small change in c_t can cause a discontinuous jump in f_t if it causes the solution of the flow program to move from one corner of the feasible region to an adjacent one.

There are, however, some convergence results that do hold under mild conditions if $E_t \to E$, $c_t \to c$, and $s_t \to s$ as $t \to \infty$. For example, if the primal and dual feasible regions are bounded and the primal and dual optima are not degenerate, then $f_t \to f$ and $v_t \to v$.

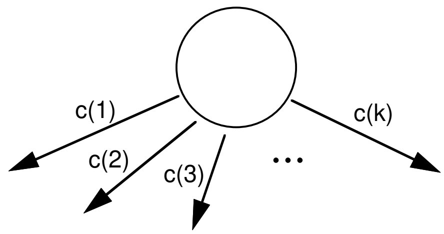


Figure 4.5: A Markov decision process with just one state.

4.4 Soft constraints

Consider the MDP shown in Figure 4.5. It has just one state; from this state the agent may choose any of k actions, with costs $c(1) \ldots c(k)$, each of which end the trajectory. The primal and dual linear programs for this MDP are

$$\text{maximize } v \text{ subject to } v \leq c(1), v \leq c(2), \ldots, v \leq c(k)$$

$$\text{minimize } c^{\mathrm{T}} f \text{ subject to } \sum_i f_i = 1, f \geq 0$$

where c is the vector with elements $c(1) \ldots c(k)$. If $c(i)$ is the smallest element of c, then the solution to the value program is $v = c(i)$, while the solution to the flow program is a vector f with a 1 in the ith position and zeros elsewhere.

Let $c_1, c_2, \ldots$ be a sequence of vectors converging to c, and let v_t and f_t be the solutions to the linear programs that result from replacing c with c_t in the value and flow programs above. Then v_t will converge to v, and f_t will converge to f as long as there is a unique smallest element of c.

Unfortunately, though, v_t may not be the best estimate of v given c_t. As pointed out in [TS93], if the elements of $c_t - c$ are random variables with zero mean, then v_t will tend to underestimate v. The reason for this behavior is that the errors in the components of c_t can cause the smallest element of c_t to have a different index than the smallest element of c. The underestimation will be most pronounced if there are several elements of c that have almost the same value as the smallest element.

We can at least partially fix this problem by "softening" the inequality constraints in the value program, so that v_t is allowed to be slightly larger than the smallest component of c_t. To do so, we will pick a penalty function l and scaling factor $\mu > 0$ and replace the constraint $v_t \leq c_{t,i}$ by the penalty $\mu l(\frac{v_t - c_{t,i}}{\mu})$. The idea is that $l(x)$ should be small for negative values of x and large for positive values of x, so that there is a penalty for making v_t too much larger than the smallest component of c_t. The scaling factor lets us specify how much uncertainty there is in the components of c_t: the smaller μ is, the faster the penalty grows as v_t increases.

More precisely, let l be any convex function with $l'(x) \to 0$ as $x \to -\infty$ and $l'(x) \to \infty$ as $x \to \infty$. As in the last chapter, l' stands for any subgradient of l. If $\operatorname{dom} \partial l$ is not all of $\mathbb{R}$, then we extend l' to $\mathbb{R}$ by taking $l'(x) = -\infty$ for x to the left of $\operatorname{dom} \partial l$ and $l'(x) = +\infty$ for x to the right of $\operatorname{dom} \partial l$. Given such a penalty function l, we define the soft value program with parameter $\mu > 0$ to be

$$\text{maximize } v - \mu \sum_i l\left(\frac{v - c(i)}{\mu}\right)$$

If we take $l(x)$ to be $\delta(x|x \leq 0)$ then the soft value program is identical to the value program for any μ. Usually, though, we will take $l(x)$ to be a function that approaches its limits more gradually, say $l(x) = e^x$. In this case the value of μ controls how hard or soft the constraints are: smaller values of μ result in harder constraints. In fact, under mild conditions the solution to the soft value program will approach the solution to the original value program as $\mu \to 0$.

The dual of the soft value program is the soft flow program

$$\text{minimize } c^{\mathrm{T}} f + \mu \sum_i l^*(f_i) \text{ subject to } \sum_i f_i = 1$$

The terms $l^*(f_i)$ serve as barriers to push the elements of f away from zero, so the constraint $f \geq 0$ is no longer necessary. (Because of this fact, μ is usually called the barrier parameter.) For example, if $l(x) = \delta(x|x \leq 0)$ then $l^*(x) = \delta(x|x \geq 0)$; or if $l(x) = e^x$ then $l^*(x) = x \ln x - x$. More generally, since $l'(x) \to 0$ as $x \to -\infty$, $l^*(x)$ will be equal to ∞ for $x < 0$, and since $l'(x) \to \infty$ as $x \to \infty$, $l^*(x)$ will be finite for any positive x.

The barrier terms tend to push the components of f away from zero, while positive components of c tend to push the corresponding components of f towards zero. So, the largest component of f will correspond to the smallest component of c. The larger μ is, the closer f will be to the uniform distribution, and the smaller μ is, the more f will concentrate its weight on the smallest components of c. In fact, just as with the soft value program, under mild conditions the solution to the soft flow program approaches the solution to the original flow program as $\mu \to 0$.

Just as in the previous section, if $c_1, c_2, \ldots$ is a sequence of vectors converging to c, then we can define an incremental algorithm for computing f or v by substituting c_t for c in the flow or value programs. (Because we have assumed a particular form for the edge matrix we do not need to reveal it incrementally, and because there is only one state we do not need to reveal the vector of starting frequencies incrementally.) We do have to make one additional choice, though: we must choose a sequence of barrier parameters μ_t converging to μ. (In particular, to solve the original linear program with hard constraints, we should have $\mu_t \to 0$.) Then we can write the incremental soft flow program as

$$\text{minimize } c_t^{\mathrm{T}} f_t + \mu_t \sum_i l^*(f_{t,i}) \text{ subject to } \sum_i f_{t,i} = 1$$

where $f_{t,i}$ is the ith component of f_t. The incremental soft value program can be written similarly.

We have been using a simple linear program as an example, but we can soften the constraints of an arbitrary linear program in the same way. To the linear program

$$\text{minimize } c^{\mathrm{T}} f \text{ subject to } E^{\mathrm{T}} f + s = 0, f \geq 0 \tag{4.5}$$

corresponds the softened program

$$\text{minimize } c^{\mathrm{T}} f + \mu \sum_i l^*(f_i) \text{ subject to } E^{\mathrm{T}} f + s = 0 \tag{4.6}$$

with its dual

$$\text{maximize } s^{\mathrm{T}} v - \mu \sum_i l\left(\frac{Ev + c}{\mu}\right)$$

Linear programs are invariant to scaling; that is, for any $k > 0$ the program

$$\text{minimize } kc^{\mathrm{T}} f \text{ subject to } kE^{\mathrm{T}} f + ks = 0, f \geq 0$$

has the same primal and dual solutions as (4.5). In order to make the soft programs invariant to scaling, we must scale μ as well; the program

$$\text{minimize } kc^{\mathrm{T}} f + k\mu \sum_i l^*(f_i) \text{ subject to } kE^{\mathrm{T}} f + ks = 0$$

has the same primal and dual solutions as (4.6).

4.5 A statistical interpretation

While there are many possible choices for the penalty function l in the soft value and flow programs, picking $l(x) = e^x$ for the MDP of Figure 4.5 results in a familiar algorithm. Since $l^*(x) = x \ln x - x$, the soft flow program can be written

$$\text{minimize } c^{\mathrm{T}} f + \mu H(f)$$

$$H(f) \stackrel{\text{def}}{=} \sum_i f_i \ln f_i + \delta\left(f | \sum_i f_i = 1\right)$$

This minimization problem is almost the same as the one that yields the WM algorithm from the previous chapter. To complete the analogy, write x_t for the vector of expert losses on trial t and let $c_t = \frac{1}{t} X_t \stackrel{\text{def}}{=} \frac{1}{t} \sum_{i=1}^{t-1} x_t$. Then if we set $\mu_t = \frac{\eta}{t}$, we have

$$f_t = \arg\min_f (c_t^{\mathrm{T}} f + \mu_t H(f)) = \arg\min_f (H(f) + \eta X_t)$$

so the incremental flow programs produce the same series of predictions as the WM algorithm.

These predictions also have a simple Bayesian interpretation. Let us suppose that the experts are predicting a sequence of binary random variables y_t. Suppose also that there is a single best expert, so that the true outcome is always equal to the best expert's prediction plus some random noise. Our task is then to distinguish between the n statistical models "expert i is best." Write $p_{1,i}$ for the prior probability that expert i is best, and $p_{t,i}$ for the posterior after seeing the first $t-1$ examples. Write $p(y|\hat{y})$ for the probability of outcome y given that the best expert predicts $\hat{y}$. Then we can compute the posterior probabilities of our models with Bayes' rule:

$$p_{t+1,i} \propto p_{t,i} p(y_t|\hat{y}_{t,i})$$

So, if we initialize $X_{0,i} = -\ln p_{0,i}$ and update

$$X_{t+1,i} = X_{t,i} + x_{t,i}$$

where $x_{t,i} = -\ln p(y_t|\hat{y}_{t,i})$, then the posterior probabilities at each time step are just $p_{t,i} = \exp(X_{t,i}) / \sum_j \exp(X_{t,j})$, which are the same as the predictions of the WM algorithm with learning rate $\eta = 1$.

More generally, we can interpret the soft value and flow programs for arbitrary Markov decision processes (or in fact any primal and dual pair of softened linear programs) as statistical estimation problems. The remainder of this section explores this connection in more detail.

4.5.1 Maximum Likelihood in Exponential Families

One of the simplest statistical inference methods is maximum likelihood: given a family of probability distributions, pick the one which maximizes the probability of an observed sample. More formally, suppose we have a set $\mathcal{X}$ of possible outcomes. (We will assume $\mathcal{X}$ is finite, but much of the following carries over to infinite sets of outcomes.) Write $\bar{f}_x$ for the normalized frequency of outcome $x \in \mathcal{X}$ in the observed sample. Suppose that our family of distributions is indexed by a parameter vector θ, and write $f_x(\theta)$ for the predicted probability of outcome x given θ. Then the maximum likelihood problem is to find

$$\arg\max_{\theta} \sum_{x \in \mathcal{X}} \bar{f}_x \ln f_x(\theta) \tag{4.7}$$

that is, to find the θ which maximizes the log-likelihood of the observed sample.

Often the distributions $f_x(\theta)$ will form an exponential family, that is, a set of distributions for which we can write

$$f_x(\theta) = \exp(t_x^{\mathrm{T}} \theta + h_x + g(\theta)) \tag{4.8}$$

Many well-known sets of distributions are exponential families, for example the normal, gamma, exponential, chi-squared, Dirichlet, multinomial, and Poisson families.

In Equation 4.8 the vectors t_x and scalars h_x, one for each possible outcome $x \in \mathcal{X}$, together define the exponential family. The function $g(\theta)$ is called the cumulant generating function, and it is determined by the requirement

$$(\forall\theta) \ \sum_{x\in\mathcal{X}} f_x(\theta) = 1 \tag{4.9}$$

We can interpret each component of t_x as a relevant feature or statistic about outcome x. For example, if $\mathcal{X}$ is a set of real numbers, we can associate the features x and x^2 with outcome x. Then we can set $h_x = 0$, and the result will be a family of discrete normal distributions. The constants h_x allow us to define subfamilies: for example, we can define a family with fixed variance by setting h_x to a multiple of x^2 and using just the single feature x.

One reason exponential families are important is that their maximum likelihood problems can be written as convex programs. By substituting (4.8) into (4.7) and using (4.9) to constrain g to be equal to $g(\theta)$, we can see that the maximum likelihood problem for an exponential family is

$$\arg\max_{\theta,g} \sum_{x\in\mathcal{X}} \bar{f}_x(t_x^{\mathrm{T}}\theta + g) \quad \text{subject to} \quad \sum_{x\in\mathcal{X}} \exp(t_x^{\mathrm{T}}\theta + h_x + g) = 1$$

The above is not a convex program, since the equality constraint does not in general define a convex set. However, it is equivalent to the convex program

$$\arg\max_{\theta,g} \sum_{x\in\mathcal{X}} \bar{f}_x(t_x^{\mathrm{T}}\theta + g) - \sum_{x\in\mathcal{X}} \exp(t_x^{\mathrm{T}}\theta + h_x + g)$$

To see why, we can explicitly perform the maximization with respect to g by differentiating and setting to 0:

$$\begin{aligned} 0 &= \sum_{x\in\mathcal{X}} \bar{f}_x - \sum_{x\in\mathcal{X}} \exp(t_x^{\mathrm{T}}\theta + h_x + g) \\ 1 &= \sum_{x\in\mathcal{X}} \exp(t_x^{\mathrm{T}}\theta + h_x + g) \end{aligned}$$

This expression is exactly the equality constraint from the maximum likelihood problem, and substituting it back into the maximization gives us the correct objective function.

4.5.2 Maximum Entropy and Duality

We can gain some insight into the maximum likelihood problem for exponential families by noticing that it is the convex dual to another problem, called linearly constrained maximum entropy. As mentioned earlier, the maximum likelihood problem for exponential families is to find

$$\arg\max_{\theta,g} \sum_{x\in\mathcal{X}} \bar{f}_x(t_x^{\mathrm{T}}\theta + g) - \sum_{x\in\mathcal{X}} \exp(t_x^{\mathrm{T}}\theta + h_x + g)$$

We can write this problem more compactly by making two slight modifications. First, if we redefine t_x by adding an extra component at the end which is always 1, then we can represent g as the last component of θ instead of writing it separately. Second, if we define a matrix T whose rows are the feature vectors t_x, we can write the problem in matrix notation. With these two modifications the problem becomes

$$\arg\max_{\theta} \bar{f}^{\mathrm{T}} T\theta - \sum \exp(T\theta + h)$$

where for any vector x the notation $\exp(x)$ means the vector whose components are $\exp(x_i)$ and $\sum x$ means $\sum_i x_i$.

The constrained maximum entropy problem makes no reference to θ or the exponential family. Instead it is defined over all probability distributions which agree with $\bar{f}$ on the expected value of each feature. Subject to these linear constraints, we wish to find the distribution which maximizes entropy with respect to some known distribution q. In other words, we want

$$\arg\min_{f \geq 0} \sum_{x \in \mathcal{X}} f_x \ln f_x - \sum_{x \in \mathcal{X}} f_x \ln q_x \quad \text{subject to} \quad T^{\mathrm{T}} f = T^{\mathrm{T}} \bar{f} \tag{4.10}$$

Since $\bar{f}$ is normalized, and since we added an extra column of 1s to T, one of the equality constraints in (4.10) forces f to be a probability distribution.

To convert maximum entropy into maximum likelihood, we need to use a vector of Lagrange multipliers (call it λ) to eliminate the equality constraints:

$$\arg\min_{f \geq 0} \max_{\lambda} \sum_{x \in \mathcal{X}} f_x \ln f_x - \sum_{x \in \mathcal{X}} f_x \ln q_x + \lambda^{\mathrm{T}}(T^{\mathrm{T}} \bar{f} - T^{\mathrm{T}} f)$$

Then we can dualize by interchanging the order of minimization and maximization and performing the minimization explicitly. Since the minimum must occur at an interior point of the region $f \geq 0$, we can find it by setting derivatives to zero:

$$\begin{aligned}
0 &= \frac{\partial}{\partial f_x} \left[\sum_{x \in \mathcal{X}} f_x \ln f_x - \sum_{x \in \mathcal{X}} f_x \ln q_x + \lambda^{\mathrm{T}}(T^{\mathrm{T}} \bar{f} - T^{\mathrm{T}} f) \right] \\
&= 1 + \ln f_x - \ln q_x - \lambda^{\mathrm{T}} t_x \\
f_x &= \exp(\lambda^{\mathrm{T}} t_x + \ln q_x - 1)
\end{aligned}$$

Substituting this value of f back into the optimization problem and cancelling terms gives (note that we have performed the substitution in two stages to make the cancellations clearer):

$$\begin{aligned}
& \arg\max_{\lambda} \sum_{x \in \mathcal{X}} \left[f_x(\lambda^{\mathrm{T}} t_x + \ln q_x - 1) - f_x \ln q_x + \lambda^{\mathrm{T}} t_x (\bar{f}_x - f_x) \right] \\
= \; & \arg\max_{\lambda} \sum_{x \in \mathcal{X}} \left[-f_x + \lambda^{\mathrm{T}} t_x \bar{f}_x \right] \\
= \; & \arg\max_{\lambda} \left[- \sum_{x \in \mathcal{X}} \exp(T\lambda + \ln q - 1) + \bar{f}^{\mathrm{T}} T\lambda \right]
\end{aligned}$$

Finally, putting $h_x = \ln q_x - 1$ and $\theta = \lambda$ completes the transformation from maximum entropy to maximum likelihood.

4.5.3 Relationship to linear programming and MDPs

We can interpret the constrained maximum entropy problem (4.10) as a linear program plus an entropy barrier term. In fact, the only differences between Equation 4.6 with $l^*(x) = x \ln x$ and Equation 4.10 are that the barrier term in Equation 4.10 has a fixed weight, the matrix T in Equation 4.10 is required to have a column of all ones, and the vector $\bar{f}$ is required to sum to 1. The first difference is not a loss of generality, since we can always rescale any soft linear program so that the barrier term has weight 1.

The required column of ones in T and the normalization of $\bar{f}$ are a loss of generality compared to Equation 4.6, but we can remove these restrictions from Equation 4.10 without damaging its statistical interpretation. Allowing an arbitrary $\bar{f} > 0$ just means that f no longer has to sum to 1; we can interpret such an f as encoding both a probability distribution and a sample size. The constant column in T serves to make the sample size of f match the observed sample size from $\bar{f}$, just as any other column of T serves to make some other feature of f match its observed value from $\bar{f}$. So, a matrix T without a constant column corresponds to a statistical estimation problem in which we have not observed the sample size. While such statistical estimation problems are unusual, they do exist. In fact, Markov decision processes are a good example: in an MDP we observe how often trajectories start at each state, but we do not observe how often we should visit each transition, since the latter depends on which policy we follow. So, the sample size (that is, the total number of transitions we visit while following an optimal policy) is just another parameter that we can estimate from the observed data. Trying to constrain the sample size of f to match that of $\bar{f}$ would be a mistake: for example, in a shortest-paths MDP this constraint would prevent us from considering exactly the policies that we want to consider, the ones that visit fewer states than our sample trajectories do.

4.6 Introducing approximation

Section 4.4 discussed how we can soften the constraints in the linear program representation of a Markov decision process. This softening combats the systematic errors introduced by random fluctuations in our estimates of the coefficients. The amount of softness is controlled by the barrier parameter μ. As we get better estimates of the coefficients, our goal is to reduce μ to zero.

In this section we will discuss how to introduce an approximate representation of the value function into the linear program for a Markov decision process. These two modifications, softening and approximation, are complementary: approximation introduces errors into the coefficients, and we can minimize the effects of these errors by a process related to softening.

$$
\begin{aligned}
&\text{maximize } v(4) \\
&\text{subject to} \\
v(0) &\leq 0 \\
v(1) - v(0) &\leq 1 \\
v(2) - v(1) &\leq 2 \\
v(3) - v(2) &\leq -1 \\
v(4) - v(3) &\leq 2
\end{aligned}
$$

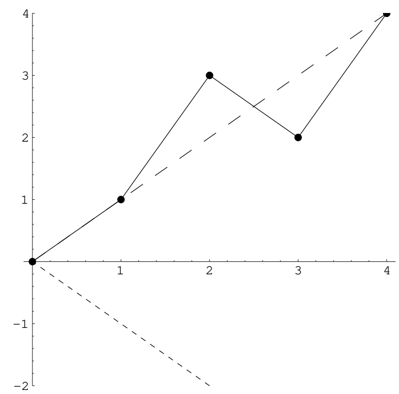


Figure 4.6: A linear program with its true solution and two approximate solutions.

4.6.1 A first try

Suppose that we have decided on a particular approximate representation for our value function, say $v = Aw$. Here w is a vector of adjustable parameters and A is a matrix whose columns are a set of basis vectors for representing v. The matrix A will have one row for each state in our MDP and one column for each basis vector. This notation encompasses any representation for v that is linear in its parameters, including linear or polynomial regression, splines, wavelets, CMACs, and many others.

The simplest way to introduce this approximate representation into our linear program is just to substitute Aw for v everywhere. Doing so yields the following modification of the value program

$$\text{maximize } s^{\mathrm{T}} Aw \text{ subject to } EAw + c \geq 0 \tag{4.11}$$

with the dual

$$\text{minimize } c^{\mathrm{T}} f \text{ subject to } A^{\mathrm{T}}(E^{\mathrm{T}} f + s) = 0, f \geq 0$$

The solution to Equation 4.11 can be a good approximation to the true value function v, particularly if the span of our basis function matrix A contains a low-error approximation to v. For examples of some MDPs for which this approach works well, see [TZ93, TZ95, TZ97].

On the other hand, if the best representations in the span of A have moderate error, then the quality of the solution we find with Equation 4.11 can degrade rapidly. For example, Figure 4.6 shows the linear program corresponding to a simple MDP, along with two approximate solutions. The true solution is shown as a solid line. If we substitute in the representation $v(x) = w_1 x + w_0$ we might hope to get the approximate solution shown in long dashes. But instead, Equation 4.11 yields the solution shown in short dashes. The reason is that the inequality $v(3) - v(2) \leq -1$ constrains the slope w_1 to be no greater than -1. The solution in long dashes violates this constraint (its Bellman residual along this edge is negative) and so is not feasible. More generally, if our basis matrix A has k columns, the solution to Equation 4.11 will satisfy the k most restrictive constraints exactly and leave the others slack. If our approximate representation for v is inflexible enough, it is even possible that Equation 4.11 will have no solutions.

To see what the problem is with Equation 4.11, we can turn to the interpretation of a linear program as a game. As Equation 4.1 shows, a linear program is a minimax problem for a bilinear form called the Lagrangian. If the linear program is

$$\text{minimize } c^{\mathrm{T}} f \text{ subject to } E^{\mathrm{T}} f + s = 0, f \geq 0$$

then the minimizing player must choose a vector s and a nonnegative vector f, while the maximizing player simultaneously chooses a vector v; then the payoff to the maximizing player is the value of the Lagrangian

$$L(f, v) = c^{\mathrm{T}} f + v^{\mathrm{T}}(E^{\mathrm{T}} f + s)$$

If we now substitute the approximate representation $v = Aw$ into this game, we have restricted the actions of the maximizing player while leaving the minimizing player untouched. In doing so we have given the minimizing player an advantage.

4.6.2 Approximating flows as well as values

To restore balance to the game, we must somehow restrict the minimizing player. We will do so by adding a penalty term $l(f)$ to the Lagrangian. The resulting penalized Lagrangian is

$$L_p(f, v) = c^{\mathrm{T}} f + v^{\mathrm{T}}(E^{\mathrm{T}} f + s) + l(f)$$

Just as before, the minimizing player wants to choose $f \geq 0$ to make $L_p(f, v)$ as small as possible, while the maximizing player simultaneously chooses v to make $L_p(f, v)$ as large as possible. There are many different possible penalty terms, each leading to a different algorithm. Depending on how we choose the penalty, the resulting game may favor the minimizing player, the maximizing player, or neither. We have already seen one example of a possible penalty, the barrier term in the soft flow program. A disadvantage of using the barrier term as our only penalty is that it is not clear how to choose the barrier parameter μ to exactly cancel the advantage we have given to the minimizing player.

For the remainder of this chapter we will examine a different kind of penalty term: we will restrict the minimizing player's choice of f to lie in a linear subspace. If the subspace is given as the span of the columns of the matrix B, then restricting f to lie in $\operatorname{span} B$ is equivalent to using the penalty term $\delta(f \,|\, \operatorname{span} B)$.

The advantage of this type of penalty term is that there is a simple way to maintain balance between the two players. If our MDP has n states and m

edges, and if the matrix A we are using to approximate the value function has rank k, then we can choose B to have rank $m-n+k$. That way we have taken $n-k$ degrees of freedom away from each player. Different choices for B will result in different algorithms.

Choosing the penalty $\delta(f|\operatorname{span} B)$ to compensate for the approximate representation $v = Aw$ results in the problem

$$\min_{f\geq 0}\max_{v}\left(c^{\mathrm{T}}f + (Aw)^{\mathrm{T}}(E^{\mathrm{T}}f+s) + \delta(f|\operatorname{span} B)\right)$$

which we can also express as the linear program

$$\text{minimize } c^{\mathrm{T}}f \text{ subject to } A^{\mathrm{T}}(E^{\mathrm{T}}f+s) = 0, f \geq 0, f = Bg \tag{4.12}$$

We will discuss algorithms for solving such a linear program in Section 4.7.

4.6.3 An analogy

It is instructive to consider an analogy to the problem of solving an overdetermined system of linear equations. Suppose we have an $n \times n$ square matrix M and an n-vector b and we want to find an x so that $Mx = b$. Suppose also that M is so large that we need to use the approximate representation $x = Ay$, where A is an $n \times k$ matrix of basis vectors. The system $MAy = b$ is overdetermined, and so in general will have no solutions.

To find a reasonable value for y, we can write the system of equations as a minimax problem:

$$\max_{x}\min_{p} p^{\mathrm{T}}(Mx-b)$$

Since we have restricted the actions of the maximizing player by requiring $x = Ay$, we need to define a penalty function $l(p)$ that restricts the actions of the minimizing player. One common choice for $l(p)$ is the squared Euclidean length of p. This choice of penalty results in the algorithm called least squares or linear regression: since $\frac{1}{2}||\cdot||_2^2$ is a self-dual function,

$$\min_{p}\left(p^{\mathrm{T}}(MAy-b) + \frac{1}{2}||p||_2^2\right) = -\frac{1}{2}||b-MAy||_2^2$$

Another choice of penalty is the indicator function $\delta(p|\operatorname{span} B)$ for some $n \times k$ matrix B. A little algebra shows that the solution to the resulting minimax problem satisfies the system of equations

$$B^{\mathrm{T}}MAy = B^{\mathrm{T}}b \tag{4.13}$$

Equation 4.13 shows why the appropriate dimensions for B are $n \times k$: if we don't take the same number of degrees of freedom away from the minimizing and maximizing players, Equation 4.13 will be either over- or underdetermined.

If we choose $B = MA$, then the equations in (4.13) are called the normal equations. The solution to the normal equations is the same as the solution to the least squares problem. The fact that we can represent linear regression in these two different ways is a consequence of the fact that the derivative of $\frac{1}{2}||p||_2^2$ is the identity function; this connection is similar to the idea of a link function described in Sections 3.6 and 3.10.

Other possible choices for B include setting $B = DA$ for some diagonal matrix of nonnegative weights D, and setting each column of B to be a different one of the n unit vectors in $\mathbb{R}^n$. The choice $B = DA$ is not used very often, since it is not usually any easier to implement than linear regression. Setting B to a collection of unit vectors is the same as picking k of the n equations in $MAy = b$ to solve and throwing the others away. This algorithm is useful since it requires much less computation than linear regression, although the quality of the resulting solution may not be as good.

When our Markov decision process is a Markov process, the linear program for finding the value function reduces to a set of linear equations. So, we can use any of the above approximate linear equation solving algorithms to find an approximation to the value function of a Markov process. Chapter 5, including Sections 5.3.4, 5.3.5, and 5.4.1, contains a more detailed comparison of these algorithms.

4.6.4 Open problems

The choice of $\delta(f|\operatorname{span} B)$ as a penalty term is not perfect. Its largest problem is that the linear program (4.12) does not necessarily have a solution: it is possible that restricting f to be in the span of B makes (4.12) infeasible, and it is possible that restricting v to be in the span of A makes (4.12) unbounded.

If we know a vector $f_0 \geq 0$ which is either feasible or approximately feasible, there is a simple trick to make sure that Equation 4.12 has a solution. If f_0 is exactly feasible we can replace whatever B we were going to use by DB, where D is the diagonal matrix with entries f_0. Then, as long as the original B could represent the vector of all ones, DB can represent f_0. If we now ensure that (4.12) is bounded, for example by requiring that the cost vector is positive ($c \geq 0$), then there will be a finite optimal solution. If, on the other hand, f_0 is only approximately feasible, we can replace the starting frequencies s by $-E^{\mathrm{T}} f_0$. If f_0 were exactly feasible then this replacement would not change the starting frequencies, since feasibility implies $-E^{\mathrm{T}} f_0 = s$. Since f_0 is not feasible, the replacement will change s so that f_0 is feasible in the modified program. Then we can set D to the diagonal matrix with entries f_0 and proceed as before.

Even if we do ensure feasibility this way, though, there is no guarantee that any vector other than f_0 is feasible. In other words, it may not be possible to evaluate any policy other than the one which generated our training data.

Another difficulty is that, while the most pleasing approximations to the value function have approximately equal total Bellman error in the positive and negative directions, the performance of the greedy policy is affected in an inherently asymmetric way by Bellman errors of opposite sign. Positive residuals correspond to states whose estimated cost is too low, and such states tend to attract flow, while negative residuals correspond to states whose estimated cost is too high, and such states tend to repel flow. So, in the worst case, a single large positive error could cause the greedy policy to spend all of its time in one state, while a single large negative error can only cause the greedy policy to avoid one state (plus any states which are only reachable through that state).

Besides the restricting the minimizing player to a linear subspace, there are many other ways to choose a penalty function. For example, we could restrict the minimizing player to a convex set such as a cube or a simplex instead of to a subspace. Or, we could remove some restrictions on the minimizing player while adding others: for example, while we have restricted the minimizing player to the intersection of the positive orthant with the span of B, we could equally well have restricted to the projection of the positive orthant onto the span of B. Finally, at the cost of giving up convexity, we could restrict the minimizing player to a nonlinear subspace. We experimented briefly with these and other approaches, but the version of the algorithm given here is the one that seemed to work best.

Yet another approach is suggested by the correspondence between the soft penalty term introduced in Section 4.4 and maximum likelihood estimation. Negative Bellman residuals in an MDP program with a soft penalty term correspond to samples in a maximum likelihood problem that have low probability under the best model. Such samples are often called outliers, under the assumption that they were generated by some process that we cannot model. In maximum likelihood estimation, two possible responses to outliers are to discard them or to add additional representational power to the model. We could apply these same principles to solving MDPs by either discarding transitions with large negative residuals or adding representational power to our model of the value function.

4.7 Implementation

The previous section outlined at a high level the choices involved in designing an algorithm to approximate the Bellman linear program for a Markov decision process. This section describes in more detail the implementation we used to perform our experiments.

4.7.1 Overview

There are several design decisions that we had to make for our algorithm. The first is how to represent our knowledge about the Markov decision process, including its dynamics, its goals, and its starting state frequencies. We chose to represent the MDP's dynamics and goals with a list of the transitions we have sampled; so, for each transition, we store its one-step cost and the feature vectors for its starting and ending states. To represent the starting frequencies, we store our estimate of the expected feature vector for a state chosen from the starting distribution.

The second decision is what representation to use for the flows. As discussed in Section 4.6.2, we want to restrict the minimizing player to a subspace of the possible flow vectors in order to counterbalance the fact that we have restricted the maximizing player to a subspace of the possible value functions. We can represent the allowable subspace of flow vectors as the span of a matrix B.

In our implementation we use the following choice for B. There is one row for each transition we have observed. The first k columns of the row contain the feature vector for the starting state of the transition. That means that the first k columns of B are a copy of A with some rows duplicated. The remaining $m-n$ columns of each row contain either one or two nonzero elements, and are used to chain together all of the actions that have the same starting state. If rows $i_1 < i_2 < \ldots < i_j$ all start from the same state, there will be a 1 in position $(k+i_1, i_1)$, a -1 in position $(k+i_1, i_2)$, a 1 in position $(k+i_2, i_2)$, a -1 in position $(k+i_2, i_3)$, and so on until a -1 in position $(k+i_{j-1}, i_j)$. This pattern of 1s and -1s for a single starting state takes up one fewer column than it does rows, and so for n states it will take up n fewer columns than rows.

To understand this choice of B, consider the example of an MDP with exactly three actions from each state. If we sort the transitions by action, then by state within action, B will have the block representation

$$\begin{pmatrix} A & I & 0 \\ A & -I & I \\ A & 0 & -I \end{pmatrix} \tag{4.14}$$

In this example, as in general, if we write $f = Bg$ then the first k components of g assign flow equally to all actions with the same starting state, while the remaining $m-n$ components of g move flow around between actions with the same starting state. As we can see from the example in (4.14), the last $m-n$ columns of B are very sparse; so, since an $m \times (m-n)$ matrix is expensive to represent we will store only the nonzero components of these columns of B.

The final decision is whether to apply the trick described in Section 4.6.4 to make sure that the linear program is feasible. We decided not to do so, since we wanted to include information about transitions that we did not follow. Under the scheme of Section 4.6.4, such transitions would receive zero weight and so would convey no information. We did not observe any problems with infeasibility, but it could still be that reweighting in this way would have improved our learning performance.

The next section describes our implementation in more detail.

4.7.2 Details

The input to our program is a description of the transitions we have sampled from the Markov decision process and the features we plan to use to approximate the value function. More specifically, if we have seen m transitions from n states and we have k features, the input will comprise the following objects (described in more detail below):

- An $n \times k$ dense matrix A.
- An $m \times k$ dense matrix EA.
- An $(m - n + k) \times m$ sparse matrix B.
- An m-vector c.
- A k-vector $A^{\mathrm{T}}s$.

By a dense matrix we mean one where we represent every element explicity, while by a sparse matrix we mean one where we represent only the nonzero elements to save space. The output of our program is a vector of parameters w representing our learned value function.

The columns of the matrix A are the basis functions we intend to use to represent the value function. In other words, at the end of the algorithm, Aw is our best estimate of the true value function. To save space, we do not represent the rows of A that correspond to states which we have not visited. Each row of A contains the values of our k features or basis functions at a single state. For example, if our observed states were the real numbers $x_1, x_2, \ldots$ and we wanted a quadratic approximation to the value function, then the rows of A would be $(1, x_1, x_1^2), (1, x_2, x_2^2), \ldots$.

The matrix EA is our best estimate of the product of the edge matrix E with the basis matrix A. To save space, we remove from E the columns corresponding to states we have not visited and the rows corresponding to transitions we have not visited. So, each row of E corresponds to a single transition we have observed: if we observe a transition from state i to state j then the corresponding row of E will have a -1 in the ith column and a γ in the jth column. If we know not just a single destination state but a probability distribution p with nonzero mass on several destination states, then the corresponding row of E will be equal to γp except that 1 will be subtracted from the ith column. So, each row of EA contains a difference between feature vectors along a transition: if we observe a transition from state i to state j, and if state i has feature vector a_i and state j has feature vector a_j, then the corresponding row of E will be $\gamma a_j - a_i$. If we know the probability distribution p over possible destination states then we can replace γa_j by its expectation under p.

The matrix B plays the role described above: we restrict the minimizing or flow player to choose a vector in the span of B. Since the first k columns of each row of B are duplicated from A, we store the indices into A instead of these columns; and since the remaining $m - n$ columns of B are sparse, we store these columns as a list of their nonzero entries.

The vector c contains the cost of each transition. The vector $A^{\mathrm{T}}s$ is the product of our basis matrix A with the vector of starting frequencies s. We can compute $-A^{\mathrm{T}}s$ as a weighted sum of the rows of EA, with the weight of each row equal to the number of times we have traversed the corresponding transition.

Once we have these inputs, the simplest way to find the coefficients of the approximate value function is to construct the linear program

$$\text{minimize } c^{\mathrm{T}} f \text{ subject to } A^{\mathrm{T}} E^{\mathrm{T}} f + A^{\mathrm{T}} s = 0, f = Bg, f \geq 0 \tag{4.15}$$

and pass it to a prepackaged linear program solver. The estimated coefficients of the value function are then the dual variables for the equality constraints $A^{\mathrm{T}} E^{\mathrm{T}} f + A^{\mathrm{T}} s = 0$ (possibly negated, depending on how the prepackaged solver defines the dual variables). This approach works well if the prepackaged solver is set up so that it does not cause too much fill-in in matrix products involving B.

To take better advantage of the sparseness in B we have implemented an interior-point barrier method linear program solver customized for linear programs of the type (4.15). Like other logarithmic barrier methods (for example [AGMX96, Van94] and many others), our implementation approximately solves a sequence of convex programs

$$\text{minimize } c^{\mathrm{T}} f - \mu \sum_i \ln f_i \text{ subject to } A^{\mathrm{T}} E^{\mathrm{T}} f + A^{\mathrm{T}} s = 0, f = Bg \tag{4.16}$$

for decreasing values of the barrier parameter μ. The barrier parameter serves a similar purpose here to the one it served in Section 4.4: it softens the constraints and makes the convex program smoother. Whereas in Section 4.4 we wanted to smooth the constraints because of uncertainty in the coefficients of the linear program, here we just want to smooth out the constraints to make the program easier to solve. So, we will start with a large value of μ, then try to track the solution to (4.16) as we decrease μ towards zero.

The set of solutions to (4.16) for all values of μ is called the central path. Figure 4.7 shows a fragment of the central path for the simple linear program

$$\text{minimize } x + y \text{ subject to } x \geq 0, y \geq 0, x + 2y \geq 1$$

As the figure shows, the central path starts out far from any of the constraint lines, then moves smoothly towards the optimal solution.

The Lagrangian for Equation 4.16 is

$$c^{\mathrm{T}} f - \mu \sum_i \ln f_i + w^{\mathrm{T}} A^{\mathrm{T}} (E^{\mathrm{T}} f + s) + z^{\mathrm{T}} (f - Bg)$$

To find the saddle point of the Lagrangian we can set its derivatives with respect to f, g, w, and z to zero. The resulting nonlinear equations are

$$\begin{aligned} 0 &= c - \mu \frac{1}{f} + EAw + z \\ 0 &= B^{\mathrm{T}} z \\ 0 &= A^{\mathrm{T}} (E^{\mathrm{T}} f + s) \\ 0 &= f - Bg \end{aligned}$$

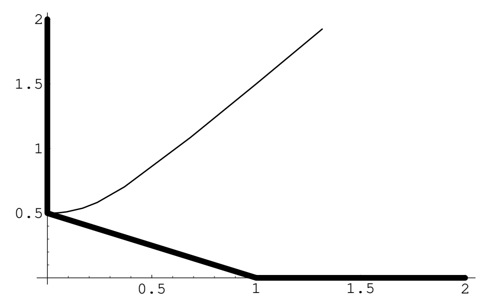


Figure 4.7: The central path for a linear program.

where $\frac{1}{f}$ means the vector whose components are the inverses of the components of f. The only nonlinearity above is in the first equation. To make the math more symmetric, we will declare a new m-vector x and replace the first equation with

$$\begin{aligned} x &= c + EAw + z \\ \mu &= f_i x_i \qquad (\forall i) \end{aligned}$$

If we have initial guesses for the variables f, g, w, z, and x, we can use Newton's method to find an update direction which brings them closer to solving Equation 4.16. That is, we can linearize the equations around our current values for the variables and solve the linearized equations for the update direction. We will require that our initial guesses for f and x are strictly positive; the updates we describe below will preserve this property.

We will linearize the equation $\mu = f_i x_i$ by replacing f_i with $f_i + \Delta f_i$ and x_i with $x_i + \Delta x_i$, then treating f_i and x_i as constants. The result is

$$\mu = f_i x_i + f_i \Delta x_i + x_i \Delta f_i + h_i$$

where h_i is the remaining higher-order term that depends on both Δx_i and Δf_i. By using the shorthand that F and X stand for the diagonal matrices with elements f and x, and that e stands for the vector of all ones, we can write the linearized equations as

$$\mu e - h - Fx = F\Delta x + X\Delta f$$

The remaining equations are already linear, but to keep the notation consistent we will replace g, w, and z by $g + \Delta g$, $w + \Delta w$, and $z + \Delta z$ and treat g, w, and z as constants. Now we can collect all of the equations into one big array:

$$\begin{pmatrix} 0 & EA & I & -I & 0 \\ A^{\mathrm{T}}E^{\mathrm{T}} & 0 & 0 & 0 & 0 \\ I & 0 & 0 & 0 & -B \\ -I & 0 & 0 & -X^{-1}F & 0 \\ 0 & 0 & -B^{\mathrm{T}} & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta f \\ \Delta w \\ \Delta z \\ \Delta x \\ \Delta g \end{pmatrix} = \ldots \tag{4.17}$$

We have avoided writing the constant on the right hand side because it would be a complicated expression and its exact form does not affect the following discussion. The matrix in (4.17) is symmetric and quasidefinite, which means that we can factor it by an algorithm similar to Cholesky decomposition; the only difference is that some of the pivots during the decomposition will be negative, so we will represent our factorization as LDL^{T} (where L is a lower triangular matrix and D is a diagonal matrix) instead of incorporating $\sqrt{D}$ into L.

Since the matrix in (4.17) is very sparse (many of its blocks are identically zero, others are diagonal, and the matrix B is sparse) we need to take care when factoring it not to introduce too much fill-in. So we will factor it partway by hand before giving it to our LDL^{T} factorizer. First we can use the fourth block row of the matrix to eliminate the fourth block column, leaving the equations

$$\begin{pmatrix} F^{-1}X & EA & I & 0 \\ A^{\mathrm{T}}E^{\mathrm{T}} & 0 & 0 & 0 \\ I & 0 & 0 & -B \\ 0 & 0 & -B^{\mathrm{T}} & 0 \end{pmatrix} \begin{pmatrix} \Delta f \\ \Delta w \\ \Delta z \\ \Delta g \end{pmatrix} = \ldots$$

This step causes no off-diagonal fill-in. Next we can eliminate the first block row and column, leaving

$$\begin{pmatrix} -A^{\mathrm{T}}E^{\mathrm{T}}X^{-1}FEA & -A^{\mathrm{T}}E^{\mathrm{T}}X^{-1}F & 0 \\ -X^{-1}FEA & -X^{-1}F & -B \\ 0 & -B^{\mathrm{T}} & 0 \end{pmatrix} \begin{pmatrix} \Delta w \\ \Delta z \\ \Delta g \end{pmatrix} = \ldots$$

This step causes some fill-in, but since EA is tall and narrow and $X^{-1}F$ is diagonal the required computation is not large. Next we will eliminate the second block row and column, leaving

$$\begin{pmatrix} 0 & A^{\mathrm{T}}E^{\mathrm{T}}B \\ B^{\mathrm{T}}EA & B^{\mathrm{T}}F^{-1}XB \end{pmatrix} \begin{pmatrix} \Delta w \\ \Delta g \end{pmatrix} = \ldots$$

Since B is sparse, we have to worry about whether this step causes fill-in. The matrix $B^{\mathrm{T}}EA$ is smaller than EA, so we don't need to worry about fill-in in this block. To analyze the block $B^{\mathrm{T}}F^{-1}XB$, first suppose that B has the form given in Equation 4.14. Then if we divide $F^{-1}X$ into a three by three block matrix with diagonal blocks D_1, D_2, and D_3, $B^{\mathrm{T}}F^{-1}XB$ is equal to

$$\begin{pmatrix} A^{\mathrm{T}}(D_1+D_2+D_3)A & A^{\mathrm{T}}(D_1-D_2) & A^{\mathrm{T}}(D_2-D_3) \\ (D_1-D_2)A & D_1+D_2 & -D_2 \\ (D_2-D_3)A & -D_2 & D_2+D_3 \end{pmatrix}$$

More generally, we can divide B into k dense columns and $m-n$ sparse columns, so $B^{\mathrm{T}}F^{-1}XB$ is of the form

$$\begin{pmatrix} \text{small and dense} & (\text{narrow and dense})^{\mathrm{T}} \\ \text{narrow and dense} & \text{large and sparse} \end{pmatrix}$$

The dot product between two different sparse columns is nonzero only if the two columns correspond to adjacent transitions from the same state, so each column of the large, sparse block has at most one nonzero element above the diagonal and at most one below. That means that we still have not caused any unacceptable fill-in.

Finally, for our last step before passing the matrix to the LDL^{T} factorizer, we can pivot along the diagonal of the large sparse block of $B^{\mathrm{T}}F^{-1}XB$. The result is a completely dense symmetric matrix with four $k \times k$ blocks. Since k, the number of basis vectors, is small, we can factorize this matrix cheaply. Using this factorization we can compute Δw and the first k components of Δg; then we can substitute backwards, undoing each of the eliminations described above, to compute the remaining components of the update direction.

Once we have the update direction vector, telling us how to change our estimates of f, g, w, z, and x to move closer to a solution of Equation 4.16, we need to decide how far to move our estimates in this direction. In other words, we need to compute a step length $\lambda \in [0,1]$ which tells us what fraction of the computed update vector to add to our estimates. Usually a step length of $\lambda = 1$ is too long, since it will cause some components of f or x to become zero or negative. So, we compute the longest step λ_0 which keeps f and x positive; then we use a step length which is the smaller of 1 and

$$.666\lambda_0 + (\beta - .666)\lambda_0^2 \tag{4.18}$$

The parameter $\beta \in (0,1)$ controls how aggressively we try to approach the boundary; we use $\beta = .99995$. The motivation for (4.18) is that the true solution has f and x nonnegative, so the farther past the constraints $f \geq 0$ and $x \geq 0$ the update vector tries to take us, the less we should believe it. Equation 4.18 produces a conservative steplength near .666 if the update vector would drive f or x far past their constraints, while it produces an aggressive steplength of β if the update direction vector brings f or x exactly to the border of the positive orthant.

The foregoing discussion describes how to update f, g, w, z, and x if we know the value of the barrier parameter μ and the higher-order term h. To estimate μ and h we use a second-order predictor-corrector method. We start by computing the update and step length for $\mu = 0$ and $h = 0$. (This update is called the predictor step.) Then we estimate the higher order term by

$$h_i = \Delta f_i \Delta x_i$$

where Δf and Δx are the predictor updates to f and x. Next we compute a target μ by a heuristic called Mehrotra's rule. Mehrotra's rule is based on the observation that in the optimal solution to Equation 4.16 we have $f^{\mathrm{T}}x = m\mu$. So, we can use $\frac{f^{\mathrm{T}}x}{m}$ as an estimate of the barrier parameter μ that produced our current values for f and x. We always want to try to lower μ; to determine how much to lower it we compute

$$\mu_0 = \frac{f^{\mathrm{T}}x}{m}$$
$$\mu_1 = \frac{(f + \lambda\Delta f)^{\mathrm{T}}(x + \lambda\Delta x)}{m}$$

where λ is the step length from the predictor step and Δf and Δx are the predictor updates to f and x. The values μ_0 and μ_1 are the estimated barrier parameters before and after the predictor step. Then we set the target barrier to be

$$\mu = \mu_1 \left(\frac{\mu_1}{\mu_0}\right)^2$$

This choice of target tries to lower μ somewhat more than the predictor step alone would have. Finally we use these values for μ and h as our estimates of the barrier parameter and higher order term to compute the actual update vector and step length. (The change between the predictor step and the actual update vector is called the corrector step.) In order to use a second-order strategy like this one we have to solve the system of equations (4.17) twice with different right-hand sides; this does not cause too much extra work since we can save the factorization from the first time and reuse it the second.

In order to completely specify our algorithm, we need to pick initial estimates for f, g, w, z, and x. We choose the very simple initialization $f_i = x_i = 1$ and $g = 0$, $w = 0$, $z = 0$.

4.8 Experiments

This section describes three experiments with the algorithm of Section 4.7. The first experiment is very simple and is just intended as a sanity check; the other two are with larger and more interesting MDPs.

4.8.1 Tiny MDP

The MDP for this experiment consists of 50 states in a line. The actions are to go one state left or right. Moving off the end of the line ends the process. The cost of each action is randomly selected before the beginning of the experiment from a normal distribution with mean 1 and variance .3, and remains fixed and deterministic thereafter.

Figure 4.8 shows the exact value function (large dots) and a quadratic approximation to it (solid line). The quadratic approximation was computed by the algorithm of Section 4.7. For comparison, Figure 4.8 also shows (dashed line) the least-squares fit to the exact value function. As we expect, the least squares fit is nearer to the exact value function than the solution from the algorithm of Section 4.7, but not by much.

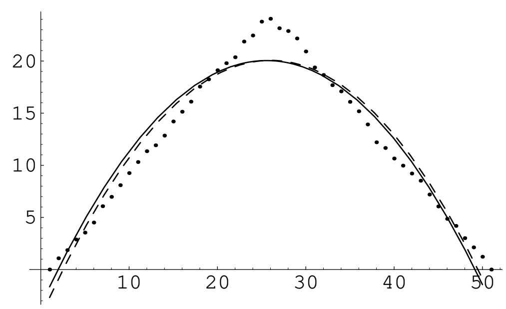


Figure 4.8: Value functions for MDP with 50 states in a line.

4.8.2 Tetris

The game of Tetris, shown in Figure 4.9, is played on a board 10 squares wide and h squares tall (we used $h = 16$). Each square of the board is either empty or full. In the space above the board the player is given one new piece at a time. Each piece consists of four filled squares arranged in one of the seven possible tetrominos (L, backwards L, S, Z, T, I, and square). Depending on which type of piece is showing, the player has up to 34 possible actions: each action consists of placing the piece in a particular orientation and horizontal position and dropping it. The edges of the piece are not allowed to extend beyond the left or right boundaries of the board. Once dropped, the piece falls straight downwards until its path is blocked by a filled square, at which point it stops moving and a new piece appears above the board. If the piece cannot move downward so that it is contained entirely within the board, the game is over. If at any point an entire row of the board is filled (that is, if there are ten horizontally adjacent filled squares) then that row disappears, the rows above it move down, and a new empty row appears at the top of the board to keep the height constant. The player scores one point for every row removed this way.

Tetris is a Markov decision process: the state consists of the arrangement of empty and filled squares (2^{10h} possibilities) and the type of piece showing (7 possibilities). The actions from each state are the possible positions and orientations from which to drop the piece. The actions have stochastic outcomes: while the motion of the piece and the scoring are deterministic, the type of the next piece is chosen uniformly at random from the possible types. We chose a discount factor of $\gamma = .99$.

The human version of Tetris has several differences. First, there are more states but fewer actions: the piece is shown moving down the board one row at a time, with enough time between downward motions to allow for several actions. The actions are to move the piece left or right one square, to turn it 90° counterclockwise, or to do nothing. Second, the human version has $h = 20$ instead of $h = 16$. Finally, the scoring for the human version is more complicated, containing bonuses for achievements such as placing pieces quickly or

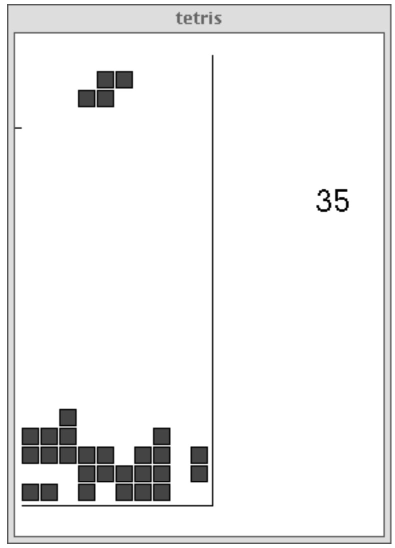


Figure 4.9: The game of Tetris.

removing several rows of filled squares at once. We chose the nonhuman version of Tetris for several reasons: except for differences in h it is the same version used in previous research [TV94, BI96]; it takes less computation per trial so our experiments can run faster; the lower height causes lower scores which also lets our experiments run faster; and it appears to be easier for the computer to learn.

We chose a very simple representation for Tetris's value function, a linear combination of just five features. All features were set to zero for game over, thus fixing the value of an ended game at zero. For a game in progress, the features were:

Constant Always equal to 1.

Average height The average height of the highest filled block in each of the ten columns.

Maximum height The maximum height of any filled block.

Airspace The total number of empty blocks that appear anywhere below a filled block in the same column.

Bumpiness The sum of the nine absolute differences between the heights of adjacent columns.

These features span a subspace of the features used in [BI96]. Although this representation is simple, it contains value functions whose greedy policies are good Tetris players: the best learned players below scored hundreds of rows in an average game.

One possible source of confusion about this representation is that it does not encode the type of the currently falling piece. This fact does not prevent a greedy policy from taking the current piece into account when it chooses an action: since the greedy policy is the result of a one-step lookahead, the current piece type affects the choice of action by determining the set of possible next states for the lookahead.

We compared the performance of two algorithms on this task. Both algorithms used the representation described above for the value function. The first algorithm was an approximate variant of policy iteration. We chose this algorithm because we believe that most researchers would accept it as a reasonable standard of comparison. In the approximate policy iteration algorithm, we played groups of five games using the same policy. After each group we ran the LS-TD(0) algorithm (described in Section 5.3.4) on that group's training examples to learn an approximate value function for the corresponding policy. After learning we switched to a new policy and threw away all previous training examples. To determine what policy to follow, we kept a running average of all of the value functions computed so far, and always acted greedily with respect to that average value function.

The second algorithm was the one described in Section 4.7. To make the comparison between the two algorithms as easy as possible, we kept as many algorithmic details as possible the same. So, we played groups of five games using the same policy, we threw away all training data every time we switched to a new policy, and we always acted greedily according to the average of all value functions computed so far. Instead of using LS-TD(0) to compute the value function after each group of games, though, we solved a linear program as described in Section 4.7. Because we kept so many algorithmic details the same for the two algorithms, we could switch between them by changing only a few lines of code.

To evaluate the performance of each algorithm we simply started it playing Tetris and recorded its scores. Figure 4.10 shows a plot of each algorithm's score as a function of how many groups of five games it had played. The plot is the average of five runs for each algorithm, and each point in a run is the average of the scores for the five games in a single group. This type of plot tends to accentuate differences between algorithms, since better algorithms will achieve longer games sooner and so will have access to more training data.

As Figure 4.10 shows, the linear programming algorithm manages to learn a decent Tetris player, but it does not achieve the performance of approximate policy iteration. Section 4.8.3 explores some possible reasons for this behavior.

We examined the weight vectors learned by the two algorithms, and they were substantially different. To check whether the difference might have been caused by slow convergence or local optima, we started the linear programming algorithm from the weight vector learned by approximate policy iteration. Within a few groups of games, the linear programming algorithm had moved away from its starting vector and back towards the answer it had converged to

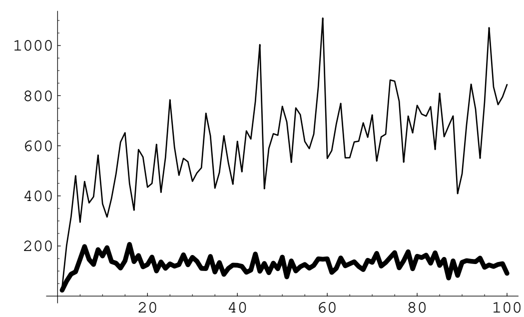


Figure 4.10: Performance of two algorithms for playing Tetris. Heavy line: linear programming. Light line: policy iteration.

from the original starting point.

4.8.3 Hill-car

We hypothesized that the linear programming algorithm's difficulty in learning Tetris was caused by trying to reason about transitions that led to states we had never visited. Since the learner has no direct constraints on the values of these states, and since their representations may be outside the convex hull of the representations of the visited states, we thought that trying to infer the values of these states might cause instability. Unfortunately, in a typical application, there is no way to avoid reasoning about unvisited states: the learner simply does not have time to explore every transition, so if we discard transitions that we have not followed, we will be reduced to a single transition out of most states.

In a small MDP, though, it is possible to visit every state. So to verify our hypothesis, we performed an experiment on a much simpler MDP. We expected that, if the unvisited states were causing our problems, the learning performance would start out poor (worse than could be explained just by lack of data), then improve rapidly as our sample size increased, and finally become acceptable once we had visited most or all of the states in the state space.

For this experiment we took the hill-car problem from Section 2.5.2, changed the time increment to $.1s$, and reduced the state space to $[-1, .7] \times [-2, 2]$ (corresponding to position$\times$velocity). Then we discretized the state space to a 20×20 grid using bilinear interpolation. The result is a 400-state, 800-edge discrete MDP. Each row of the edge matrix for this MDP has up to five nonzero entries: one negative entry for the state at time t, and up to four positive entries for the possible states at time $t + 1$.

We collected data by following a fixed policy: always thrust right. Since the goal is to get to any position greater than .6, this policy is optimal from any state where the car has sufficient momentum to reach the top of the hill, but will never terminate if the car does not start out with enough momentum. To

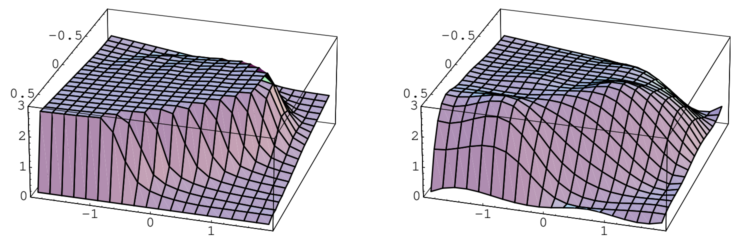


Figure 4.11: The exact value function for the hill-car MDP and a spline approximation to it.

avoid infinite trajectories, we terminated a trajectory with probability .01 on each time step, corresponding to a discount factor of $\gamma = .99$.

We represented the value function by storing the estimated values at 49 states on a 7×7 grid, then interpolating in each direction with a cubic spline. In other words, we used 49 basis functions, each of which was the product of a cubic spline depending only on position with a cubic spline depending only on velocity.

In each run of our experiment we collected seven trajectories at a time, then fed all of the trajectories so far to our learning algorithm. During each trajectory we recorded all available actions from each state we visited. We never changed policies, and we never threw away any data. After each invocation of the learning algorithm, we recorded both whether it converged and what weight vector it converged to. We collected twenty groups of trajectories, for a total of 140 trajectories per run.

We ran the experiment five times. Each time, after we had collected all 140 trajectories, the learning algorithm was able to find a good approximation to the true value function. Figure 4.11 shows the true value function and a typical approximation to it.

In three of the five runs, though, the linear programming algorithm did not converge within 75 iterations of the interior-point method when given data from only the first group of seven trajectories. In one of these three, it also did not converge when given data from the first two groups. In fact, on all runs, the linear program shows signs of ill-conditioning when data are scarce, either by lack of convergence or by convergence to an answer with a large 2-norm. Figure 4.12 shows a typical example of the latter. The value function shown in the figure is based on data from three groups of trajectories; notice that the estimated values are off of the plot scale at two corners of the state space, $(.6, -2)$ and $(-1, 2)$, where the data are particularly sparse. The full range of this learned value function is $[-52.23, 15.22]$.

On the other hand, all runs converged consistently to value functions with about the right two-norm after they had seen at least fifteen groups of seven trajectories. We believe that this behavior supports our hypothesis that transitions ending in unvisited states tend to cause instability.

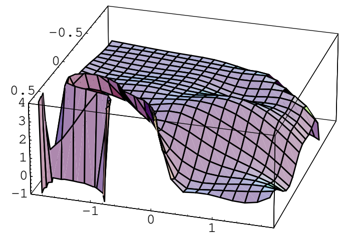


Figure 4.12: A value function learned from sparse data.

4.9 Discussion

In this chapter we have examined the connections among Markov decision processes, linear and convex programming, and maximum likelihood. Based on our analysis we have recommended a method for designing value-function approximating algorithms: substitute an approximate representation for the value function into the Bellman linear program, then add a penalty term to the dual of the Bellman program. We have coded a fast implementation of one such algorithm, and experimented with this implementation. While the learning performance of this algorithm does not improve on the best prior algorithms, we hope that the intuition and design methodology of this chapter can aid in the design of other algorithms for solving MDPs.

Chapter 5

RELATED WORK

This chapter is a brief summary of related reading on Markov decision processes. It starts by considering methods for solving small MDPs exactly, such as value iteration, policy iteration, and linear programming. Next it discusses exact methods for solving special cases of MDPs like linear-quadratic-Gaussian processes and continuous-time problems that are linear in their controls. Then it considers a variety of methods for approximating value functions. These methods range from simple interpolation on a regular grid to neural networks trained by gradient descent. Finally it describes incremental algorithms for solving MDPs.

5.1 Discrete problems

One of the first algorithms for solving Markov decision process was the Bellman-Ford single-destination shortest paths algorithm [Bel58, FF62], which learns paths in a graph (*i.e.*, a deterministic undiscounted MDP) by repeatedly updating the estimated distance-to-goal for each node based on the distances for its neighbors. The Bellman-Ford algorithm is a special case of value iteration, which is defined in Chapter 1. For other early work on similar algorithms see [Bel61, Bla65].

Besides value iteration, another good way to solve small MDPs is policy iteration. Policy iteration maintains a current policy $\pi^{(i)}$ on each step i. It solves the equation

$$v = T_{\pi^{(i)}} v$$

on each step, setting $v^{(i+1)}$ to be the solution, and then computes $\pi^{(i+1)}$ to be the greedy policy for $v^{(i+1)}$. Policy iteration often takes many fewer steps to converge than value iteration, but each step requires more work. For a proof of the convergence of policy iteration see [BT89].

Midway between value iteration and policy iteration lies modified policy iteration. In MPI we store both a current value function $v^{(i)}$ and a current policy $\pi^{(i)}$. On each step we compute the next value function $v^{(i+1)}$ from $v^{(i)}$ by the backup operator for the current policy, $v^{(i+1)} = T_{\pi^{(i)}} v_i$. On some steps we set $\pi^{(i+1)}$ to be the greedy policy for $v^{(i+1)}$, as value iteration does, but on other steps we just keep $\pi^{(i+1)} = \pi^{(i)}$. The relative frequency of these two types of step is a parameter of the algorithm: if we always choose the greedy policy, MPI reduces to value iteration, while if we usually keep the policy from the previous step, MPI behaves more like policy iteration.

Even more generally, we could store separately a value and an action for each state, and on each step improve some of the values (by setting them to the result of the value backup operator for the current policy) and some of the actions (by setting them to the action for the greedy policy). By choosing an order of updates we can produce the value iteration algorithm, the modified policy iteration algorithm, and other algorithms in between. As long as we update all actions and values often enough, the resulting algorithm converges (see [BT89]).

Finally, we can solve small MDPs by converting them to linear programs as described in Chapter 4, then solving the linear programs with simplex, barrier methods, or other linear programming algorithms [Ber76, p248] [Ros83, p40]. An MDP which takes a long time to solve by value iteration can sometimes take much less time to solve by linear programming, and vice versa. See [TZ93, TZ95, TZ97] for some comparisons between linear programming and value iteration.

5.2 Continuous problems

In the previous section we described several ways to find the exact value function for a sufficiently small, discrete MDP. None of the methods of the previous section is appropriate for solving MDPs with continuous state spaces. To solve such an MDP, we must turn either to special cases or to approximate methods.

Approximate methods for solving continuous MDPs are similar to approximate methods for solving large, discrete MDPs, so we will put off discussing them until Section 5.3. The rest of this section describes some special cases of MDPs with continuous state spaces that we know how to solve exactly.

5.2.1 Linear-Quadratic-Gaussian MDPs

One well-studied special case of continuous Markov decision processes is the linear-quadratic-Gaussian problem, where the transition function is linear in the states and controls, the cost function is quadratic, and all noise is Gaussian additive. The value function for an LQG problem is always quadratic, with coefficients given by a set of linear equations called the Ricatti equations; so, we can solve even high-dimensional LQG problems easily. (In fact, hidden state makes LQG problems only slightly more difficult.)

Even if a problem does not appear linear at first glance, it is sometimes possible to make it linear by a transformation of the state and control variables. Problems which may be so transformed are called *feedback linearizable* (see [SL91] for more detail). One important example of a feedback-linearizable model is an idealized multi-link robot arm; for this model, feedback linearization is often called the "method of computed torques." Of course, if the original model contains errors (for example, friction or backlash in the robot arm), so will the linearized model. In fact, the errors in the linearized model can be worse, since the computed control input may need to be very large to cancel the original model's nonlinearities. Another possible source of problems is that quadratic costs and Gaussian errors may no longer be quadratic and Gaussian after the transformation.

5.2.2 Continuous time

Many MDPs with continuous state spaces evolve in continuous time rather than in discrete steps. For such an MDP it is natural to write the value function as the solution to a differential equation. To do so, we must make some assumptions, the most important of which is that the MDP is deterministic.

If the state $x(t)$ of an MDP evolves according to

$$\frac{dx}{dt} = f(x, u)$$

where $u(t)$ is the control input, and if the cost of a path $x(t)$ under control $u(t)$ is $\int c(x(t), u(t))dt$, then the value function satisfies the differential equation

$$\min_u \left[\left(\frac{d}{dx} v(x) \right) \cdot f(x, u) + c(x, u) \right] = 0 \tag{5.1}$$

Equation 5.1 is called the steady-state Hamilton-Jacobi-Bellman or HJB equation [Ber95]. To ensure that the HJB equation has a unique solution, we must specify sufficiently many boundary conditions.

For some MDPs we may be able to solve the HJB equations analytically. It is easiest to solve the HJB equations analytically for Markov processes: since Markov processes allow only one choice of control u, the minimization over u is unnecessary and so the HJB equations are linear.

The paper [MM98] describes how to use a subset of averagers called barycentric interpolators to solve continuous-time Markov decision processes. The essential feature is that the authors add a requirement to the averager which ensures that, as the representational power of the averager grows, the fixed point of fitted value iteration converges to the true value function.

The following section describes a different approach to finding the best control for a continuous-time MDP.

5.2.3 Linearity in controls

Consider the single-input, single-output, nth order system

$$\left(\frac{d}{dt} \right)^n x = a(\vec{x}) + b(\vec{x})u$$

where $\vec{x}$ is a vector whose components are x and its time derivatives up to order $n-1$, a and b are (possibly nonlinear) functions of $\vec{x}$, and $b(\vec{x})$ is bounded away from zero. (For a generalization of the contents of this section to systems with k inputs and k independent outputs, see for example [SL91].)

Our goal in this section will be to supply an input $u(t)$ so that the output $x(t)$ tracks a given reference signal $x_d(t)$ as closely as possible. This goal is less general than controlling an arbitrary MDP in four important ways: first, we have replaced a general cost function by the simpler objective of tracking a known reference signal. Second, we have assumed that the system to be controlled is deterministic. Third, we have assumed that the system is linear in the control. Fourth, we have assumed that the number of control inputs is equal to the number of independent outputs. The last two assumptions in particular are often unrealistic, since they allow us to cancel an arbitrary drift by choosing a sufficiently large control input. For example, the third assumption is violated by a robot whose actuators can only exert a bounded amount of force (but see [YSS97] for a treatment of linear systems with bounded controls); the fourth is violated by the well-known cart-pole problem, which is to control the angle of a pole and the position of its base (two independent outputs) by exerting a horizontal force at the base (one input).

One benefit of making these simplifying assumptions is that we will be able to derive a controller which succeeds even if we replace a and b by estimates $\hat{a}$ and $\hat{b}$ with bounded error. (Such a controller is called *robust.*) These estimates might come, for example, from a supervised learner trained on observed system trajectories.

Before we consider robust control, we will derive a controller for use when we know a and b exactly. To that end, write $e = x - x_d$ for our tracking error, and let

$$s = \left(\frac{d}{dt} + \omega\right)^{n-1} e$$

where ω is a positive constant. The combined error measure s is a linear combination of our tracking error and its derivatives; its importance is that, if we manage to achieve $s = 0$, the tracking error e must converge exponentially to zero. To see why, consider the solutions of the differential equation $(\frac{d}{dt} + \omega)^{n-1} e = 0$. The polynomial $(x + \omega)^{n-1}$ has all of its roots at $-\omega$. So, the norm of any solution $(e, \frac{de}{dt}, \ldots, \frac{d^{n-1}e}{dt^{n-1}})$ must behave like $\exp(-\omega t)$. Since $\omega > 0$, this means that the solutions all decay to zero with time constant $\frac{1}{\omega}$.

If we define r so that $s = \frac{d^{n-1}e}{dt^{n-1}} + r$, we can solve for u in terms of $\frac{ds}{dt}$ and known quantities:

$$\begin{aligned}
\frac{ds}{dt} &= \left(\frac{d}{dt}\right)^n (x - x_d) + \frac{dr}{dt} \\
&= a(\vec{x}) + b(\vec{x})u - \left(\frac{d}{dt}\right)^n x_d + \frac{dr}{dt} \\
u &= \frac{1}{b(\vec{x})}\left(\left(\frac{d}{dt}\right)^n x_d - \frac{dr}{dt} + \frac{ds}{dt} - a(\vec{x})\right)
\end{aligned}$$

If we start out with the combined error measure s at zero, we can find the u which maintains $s = 0$ by setting $\frac{ds}{dt} = 0$ in the above equation. More generally, if $s \neq 0$, we can use a simple PD controller to reduce s by setting $\frac{ds}{dt} = -ks$ for some positive constant k. The resulting u will cause s to decay exponentially to zero.

For example, suppose we want to control the system $\ddot{x} = \cos(\exp x)$ to track $\sin t$ starting from $x = \dot{x} = 0$. If we choose $\omega = 1$, the combined error measure is $s = \dot{e} + e = (\dot{x} - \cos t) + (x - \sin t)$, and the recommended control input is $-\sin t - (\dot{x} - \cos t) - ks - \cos(\exp x)$. If we choose $k = 1$, we get

$$u = 2\cos t - 2\dot{x} - x - \cos(\exp x)$$

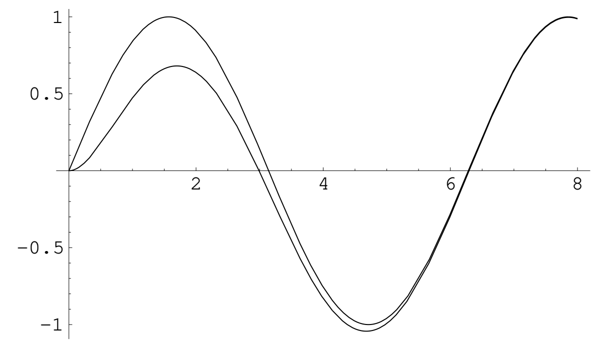


Figure 5.1: Tracking performance. The two curves are $x(t)$ and $x_d(t) = \sin t$.

Figure 5.1 shows the resulting tracking performance. Increasing either k or ω would cause faster tracking convergence at the cost of increased control activity.

The recommended control u depends on x_d and its derivatives as well as x and its derivatives. We assume that x and its derivatives can be either directly observed or computed. If the derivatives of x_d are not available (as might be the case for example if the desired trajectory were specified by a user with a joystick), a possible fix is *model-reference control.* In model-reference control, the object is to track not x_d but a filtered version of x_d. The filter is called a *reference model*, and its purpose is twofold: first to ensure that sufficiently many derivatives of the filtered x_d are available, and second to ensure that the filtered x_d is smooth enough that it can be tracked without unduly large control inputs. One common choice of reference model is a low-pass filter. More generally, the reference model might take some input other than x_d (for example derivatives of x_d) to produce the filtered x_d.

Now suppose that, instead of the exact model a and b, we have $\hat{a}$ and $\hat{b}$ instead, with $-\alpha \leq a - \hat{a} \leq \alpha$ and $\frac{1}{\beta} \leq \frac{\hat{b}}{b} \leq \beta$. (The uncertainty bounds $\alpha > 0$ and $\beta > 1$ might in general depend on $\vec{x}$ and t.) Notice that this model of uncertainty assumes that the sign of b is known, which might not be a plausible assumption in some domains.

With an uncertain model, the PD controller $\frac{ds}{dt} = -ks$ may no longer work. So, we will use instead the bang-bang control law $\frac{ds}{dt} = -k\,\mathrm{sgn}(s)$. The resulting choice of u is called a *sliding mode control.* If we choose k large enough, we can guarantee that the sliding mode controller will cause s to converge to zero even without knowing the exact a and b. (With a small amount of algebra, we can show that $k = \beta(\alpha + \eta) + (\beta - 1)|u_0|$ is large enough, where η is a small positive number and u_0 is the control that would result from setting $\frac{ds}{dt}$ to zero.) Better approximations for a and b will allow us to reduce k and use a smaller bang-bang term.

Because of the bang-bang term $-k\,\mathrm{sgn}(s)$, the sliding mode control is discontinuous in $\vec{x}$ across the surface $s = 0$. In fact, once the state hits $s = 0$, the recommended control $u(t)$ will generally have infinitely many discontinuities in any finite-length time interval. Such a control is usually physically impossible to implement; so, in practice, one would generally interpolate $u(\vec{x})$ across a thin boundary layer $-\epsilon \leq s \leq \epsilon$.

5.3 Approximation

All of the above methods are designed to find exact solutions to Markov decision processes. Because of this fact, they are usually limited to solving small or special-case MDPs. On the other hand, it is perfectly possible to run similar algorithms on an approximate representation of the solution to a decision problem. For example, Bellman discusses finding approximate value functions by quantization and low-order polynomial interpolation in [Bel61], and decomposition by orthogonal functions in [BD59, BKK63]. These approximate methods are not covered by the convergence proofs for the exact methods. But, if they do converge, they can allow us to find numerical solutions to problems which would otherwise be too large to solve.

Researchers have experimented with a number of approximate algorithms for finding value functions. Results have been mixed: there have been notable successes, including Samuels' checkers player [Sam59] and Tesauro's backgammon player [Tes90]. But these algorithms are notoriously unstable; Boyan and Moore list several embarrassingly simple situations where popular algorithms fail miserably [BM95]. Some possible reasons for these failures are given in [TS93, Sab93].

The remainder of this section discusses approximate algorithms for solving MDPs. Many of these algorithms are modifications of the exact algorithms described in Section 5.1.

5.3.1 State aggregation

The most straightforward way to approximate a continuous MDP, and one of the best-known, is to discretize the state space into a grid and assign the same value to every state in a given cell. Similarly, to approximate a large discrete MDP, we can divide the states into bins and assign the same value to every state in a given bin. For either a continuous or a discrete MDP we can then pick one sample state from each bin and run value iteration as if our samples were the entire state space. This algorithm is a special case of fitted value iteration, and so has convergence and error guarantees (see Chapter 2). State aggregation has been in use at least since the 1950s [Bel61, p86]. It is still in use today, often in combination with adaptive methods for determining how finely to discretize the state space [CT89, Moo94].

If we choose to divide each axis of a d-dimensional continuous state space into k partitions, we will wind up with k^d states in our discretization. Unfortunately, even if we choose a smallish value for k we can wind up with a huge number of states: for example, if we choose $k = 100$, a six-dimensional continuous MDP will translate into a 10^{12}-state discrete MDP. This problem is called the curse of dimensionality, since the number of states in the discretization is exponential in d.

5.3.2 Interpolated value iteration

Another important special case of fitted value iteration, dating back at least to Bellman's work in the 1950s [Bel61, p86], is the class of interpolating methods. These methods store the value function only at a predetermined set of states; when the value of some other point is needed for a backup, it is estimated by some kind of interpolation scheme. The most common schemes are to store the values of states at the vertices of a regular grid and approximate the values of other states with either constant interpolation (in which the value over an entire grid cell is the same) or multilinear interpolation. Higher-order polynomial interpolation is also possible, but can result in divergence.

For a long time, grid-based methods with constant interpolation were the only approximate variant of value iteration that was known to converge. The papers [Gor95a, TV94] were, as far as we know, the first to extend the proofs to cover even multilinear interpolation, an important extension since better interpolation methods allow us to use coarser grids and so solve larger problems. (Davies gives a two-dimensional example where piecewise constant interpolation needs about $301^2 = 90601$ cells to achieve the same level of performance as bilinear interpolation with $11^2 = 121$ cells [Dav96].)

5.3.3 Linear programming

The most straightforward way to introduce approximation into the linear program representation of the Bellman equations is simply to substitute in an approximate representation for the value function. This approach can work well, particularly if we can represent a low-error approximation of the value function. For examples of MDPs that we can solve this way see [TZ93, TZ95, TZ97].

This approach has one important disadvantage. Because we cannot represent the true value function exactly, we will not be able to satisfy the Bellman equations exactly. So, we will have to settle for some errors, that is, states whose assigned values are not equal to the backed up values from their neighbors. But, because linear programs do not allow their constraints to be violated, all of the errors in the linear-programming version of the Bellman equations will have the same sign. To put it another way, the best approximation to v^* will trade infeasibility against suboptimality, while the definition of linear programming treats feasibility and optimality asymmetrically.

Chapter 4 discusses in more detail the problem of finding an approximate value function by linear programming.

5.3.4 Least squares

For a Markov process, the Bellman equations reduce to

$$Ev + c = 0$$

where E and c are the edge adjacency matrix and cost vector for our process. E is equal to $P - I$ where P is the transition probability matrix for our process. See Chapter 1 for more detail.

We can replace v in the Bellman equations by an approximate representation, say $v = Aw$. Here A is a matrix whose columns are basis vectors for representing v, and w is a vector of adjustable parameters. If there are n states in our Markov process and we use k basis vectors to represent v, then A will be $n \times k$. With this substitution, the Bellman equations become $EAw + c = 0$. This is a system of n equations in k variables. Since in general $k < n$ (that is, since in general we use fewer basis vectors to represent v than there are states in our Markov process), these equations are overdetermined; so, they usually do not have a solution.

There are several ways to find a reasonable coefficient vector w in this situation. The simplest is to pick k of the n equations and throw away the rest. The next simplest is to choose w as the least-squares solution, that is,

$$(EA)^{\mathrm{T}} EAw + (EA)^{\mathrm{T}} c = 0$$

The vector $EAw + c$ is called the Bellman error or residual, so the least-squares solution is the one that minimizes sum of squared Bellman errors. Finally and most generally we might pick an arbitrary $n \times k$ matrix B and set

$$B^{\mathrm{T}} EAw + B^{\mathrm{T}} c = 0 \tag{5.2}$$

If we pick $B = EA$, this method reduces to least squares; or, we can define a B that keeps k of the n equations and throws away the rest by making the columns of B be k of the n unit vectors in $\mathbb{R}^n$.

One other choice for B that seems to work well is $B = DA$ for some diagonal matrix of nonnegative weights D. In particular we can set the diagonal elements of D to be the state visitation frequencies f given by

$$E^{\mathrm{T}} f + s = 0$$

where s is the vector of frequencies of starting in each state. The resulting equations are

$$A^{\mathrm{T}} \operatorname{Diag}(f)[EA + c] = 0 \tag{5.3}$$

This choice of B was popularized by the TD(0) algorithm described below in Section 5.4.1; as is explained in more detail there, TD(0) uses this choice for B because it is possible to compute an unbiased estimate of the coefficients of Equation 5.3 by sampling trajectories from the Markov process.

TD(0) never represents Equation 5.3 explicitly, but instead solves it by stochastic gradient descent. The algorithm which represents and solves Equation 5.3 explicitly is called LS-TD, for Least-Squares TD, even though it is not actually a least squares algorithm. It is described in [BB96].

Methods based on solving Equation 5.2 have an important advantage over fitted value iteration. As mentioned in Chapter 2, fitted value iteration applies a function approximator over and over again to the same value function, possibly resulting in loss of accuracy. Rather than approximating the value function directly, Equation 5.2 approximates the update direction instead. That is, while fitted value iteration computes a target value function Tv and approximates that, Equation 5.2 computes the direction from the current value function to the target value function, $(T - I)v$, and approximates that instead.

To see why this difference is important, consider the case where we are lucky enough that our function approximator can represent the optimal value function v^* perfectly. (The results will be similar if we can only represent something close to v^*.) We pointed out in Chapter 2 that fitted value iteration can still drift away from v^* if we are using the wrong kind of function approximator. On the other hand, the update direction from v^* is by definition the zero vector, and any linear function approximator can fit the zero function exactly. So, v^* will be a solution to Equation 5.2.

Unfortunately, it is difficult to generalize Equation 5.2 to find approximate solutions to Markov decision processes: since the Bellman equations for MDPs are nonlinear, it is not even clear how to decide what rank B to use to ensure that there exists a solution.

5.3.5 Collocation and Galkerin methods

Sometimes we can solve the Hamilton-Jacobi-Bellman equations approximately by numerical methods. This section describes two related techniques for doing so. These techniques work best when the HJB equations are linear, that is, for Markov processes instead of MDPs. In fact, they are in some sense the continuous time analogs of the methods in Section 5.3.4.

Suppose we wish to solve a system of differential equations numerically—say for example

$$\begin{aligned} \partial_t f(t) + f(t) &= 0 \\ f(0) &= 1 \end{aligned}$$

We begin by assuming a simple form for $f(t)$, say $f(t) = a+bt+ct^2$, and imposing the boundary constraint $f(0) = 1$ to find $a = 1$. Now we can analytically evaluate the derivative to get

$$(b + 2ct) + (1 + bt + ct^2) = 0 \tag{5.4}$$

For any given value of t, this is an ordinary algebraic equation. In fact, since the both the original differential equation and our approximation to f are linear, the algebraic equation is linear in b and c for each t. In general it will be impossible to satisfy the equation for all t, since we have replaced an arbitrary smooth function f by an approximation with only a finite number of degrees of freedom. So, we will need to pick a reduced set of equations to satisfy.

There are several ways to pick a reduced set of equations. The simplest is collocation [GO77], in which we choose just enough values of t from the interval of interest to guarantee a unique solution. In our example we have two free parameters; so, since each collocation point gives us one new equation, we need

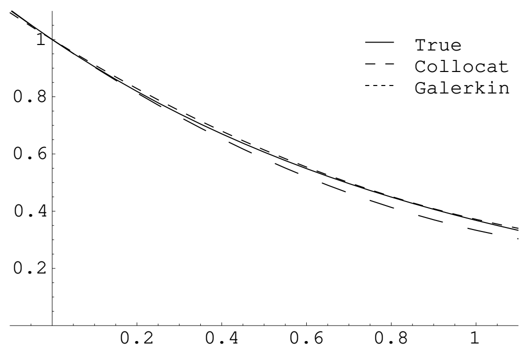


Figure 5.2: The solution to $f' + f = 0$ along with two approximations.

to collocate at two points. If we choose $t = 0, 1$, we can solve for the coefficients $b = -1, c = \frac{1}{3}$; figure 5.2 compares the resulting approximation to the true solution e^{-t} near the collocation points.

The choice of collocation points can influence the quality of our result. We can reduce the dependence of the answer on our exact choice of points, and so sometimes get a more accurate approximation, by choosing more collocation points than are strictly necessary and solving the resulting overdetermined set of equations by least squares.

Rather than a set of test points, the so-called Galerkin methods [GO77] use a set of test functions instead. Each test function specifies a weighted average of the equations for different values of t. For example, if we choose the test functions t and t^2 over the interval $[0, 1]$, Equation 5.4 yields the constraints

$$\begin{aligned} \int_0^1 t(1 + b + (b + 2c)t + ct^2)dt &= 0 \\ \int_0^1 t^2(1 + b + (b + 2c)t + ct^2)dt &= 0 \end{aligned}$$

which we can solve to find $b = -\frac{32}{35}, c = \frac{2}{7}$ (see figure 5.2).

Galerkin methods are more general than collocation, since we can choose Dirac δ-functions as our test functions in a Galerkin method and reduce it to collocation. Just as in collocation, the choice of test functions influences the quality of the resulting approximation; in our example, we have followed common practice and chosen the basis functions themselves as test functions (recall that we fixed the coefficient of 1 to satisfy the boundary condition, thus removing it from the basis).

We can use collocation or Galerkin methods to find the value function of a continuous-time deterministic Markov chain. If we assume that the goal state is at the origin, and if the state vector evolves according to

$$\frac{dx}{dt} = f(x)$$

then the value function satisfies the HJB equations

$$\begin{aligned}\left(\frac{d}{dx}v(x)\right)\cdot f(x)+c(x) &= 0\\ v(0) &= 0\end{aligned}$$

Now suppose that we choose a set of basis functions $\beta_i(x)$, each with $\beta_i(0)=0$, and perform a Galerkin approximation using the basis functions as test functions. A typical constraint will look like

$$\int_S \beta_j(x)\left[\sum_i w_i\left(\frac{d}{dx}\beta_i(x)\right)\cdot f(x)+c(x)\right]dx=0 \tag{5.5}$$

where S is the state space and w_i is the weight for β_i. While this expression looks formidable, it is actually completely analogous to the unweighted TD equations

$$A^{\mathrm{T}}[(P-I)Aw+c]=0 \tag{5.6}$$

(Equation 5.6 is the same as Equation 5.2 with the choice $B=A$.) Replacing v by Aw is analogous to replacing $v(x)$ by $\sum_i w_i\beta_i(x)$, with the ith column of A playing the same role as β_i. The term $\sum_i w_i(\frac{d}{dx}\beta_i(x))\cdot f(x)$ is the rate at which the value of the current state changes with time, given that we are in state x; it is analogous to a single component of the vector $(P-I)Aw$ in the TD equations. So, the term in square brackets in Equation 5.5 is analogous to the term in square brackets in Equation 5.6. Finally, the integral is the continuous equivalent of a dot product, so using the β_i as test functions in Equation 5.5 is analogous to the multiplication by A^{T} in Equation 5.6. The end result in either case is the same: we are computing for each state the rate of change of the value function with time, and constraining the resulting vector to be perpendicular to each of our basis functions.

Unfortunately, just as in the Section 5.3.4, it is not clear how to generalize collocation and Galerkin methods from Markov chains to Markov decision processes. Since the HJB equation is in general nonlinear, collocation or Galerkin methods will yield a set of nonlinear algebraic equations. It can be arbitrarily difficult to solve these equations; in fact it is not even clear how many collocation points or test functions are necessary to ensure that they have a unique solution.

5.3.6 Squared Bellman error

In Section 5.3.4 we discussed substituting an approximate representation for the value function into the Bellman equations. In that section, we used a representation which was linear in its parameters and we restricted attention to Markov processes; the result was that we derived a system of linear equations for the coefficients in our approximation.

In this section we will examine the more general case where we allow nonlinear function approximators such as neural networks, and where we replace

Markov processes by Markov decision processes. In this case, of course, we will not be able to find a closed-form solution for the parameters of our approximation to the value function. Instead, we will need to rely on numerical methods.

In particular, we will focus on numerical methods for finding a local minimum of the sum of squared Bellman error. The Bellman error vector for an approximate value function v is defined to be $Tv - v$, where T is the parallel value backup operator for our Markov decision process. So, the sum of squared Bellman errors is a nonnegative real-valued function of the parameters of our approximation to the value function.

Unfortunately, squared Bellman error is a badly-behaved function: it is poorly conditioned and it has derivative discontinuities. Ill-conditioning happens because the values of two states can be strongly linked even if they are separated by many time steps. (Two states will be linked when the current policy causes the agent to move from one to the other with high probability.) If we update the values of such a pair of states in opposite directions, the Bellman error will change much more quickly than if we update them in the same direction. This lack of condition means that the contours of equal Bellman error are long and narrow, so that simple minimization algorithms like gradient descent will be forced to take short steps and converge slowly.

On the other hand, methods which are more robust to ill-conditioning, such as conjugate gradient and Newton's method, often depend on the smoothness of the function to be minimized. Unfortunately, the Bellman error function can have discontinuous derivatives even for linearly-parameterized families of value functions: there will usually be a derivative discontinuity at every value function for which there is more than one greedy policy. So, for example, conjugate gradient can get caught against a derivative discontinuity in such a way that none of its line searches ever makes progress, while Newton's method can oscillate forever by stepping back and forth across a discontinuity. (Interestingly, Newton's method for minimizing $||Tv - v||_2^2$ with respect to v is identical to policy iteration, so it is guaranteed to converge; Newton's method can only have problems when we substitute an approximation for v.)

Figure 5.3 shows several views of the Bellman error surface for a very simple MDP. On the bottom row of the figure is the MDP. It has two states, so its value function is an element of $\mathbb{R}^2$: the two coordiantes are x and y, the estimated values for the left-hand and right-hand states respectively. The top row of the figure shows a 3D and a contour plot of the MDP's error surface: the x and y axes represent our current estimate of the value function, while the z axis shows the sum of squared Bellman errors for each estimate. These plots clearly show the derivative discontinuity that happens when the two actions from the right-hand state have the same backed-up value. They also show that the contours of the error surface near the global minimum can be elliptical. In this plot the ellipses are close to circular and therefore well-conditioned, but changing the transition probabilities can give the contours arbitrarily bad aspect ratios. Finally, the middle row of the plot shows the error surfaces for two different one-dimensional slices of the set of possible value functions. These one-

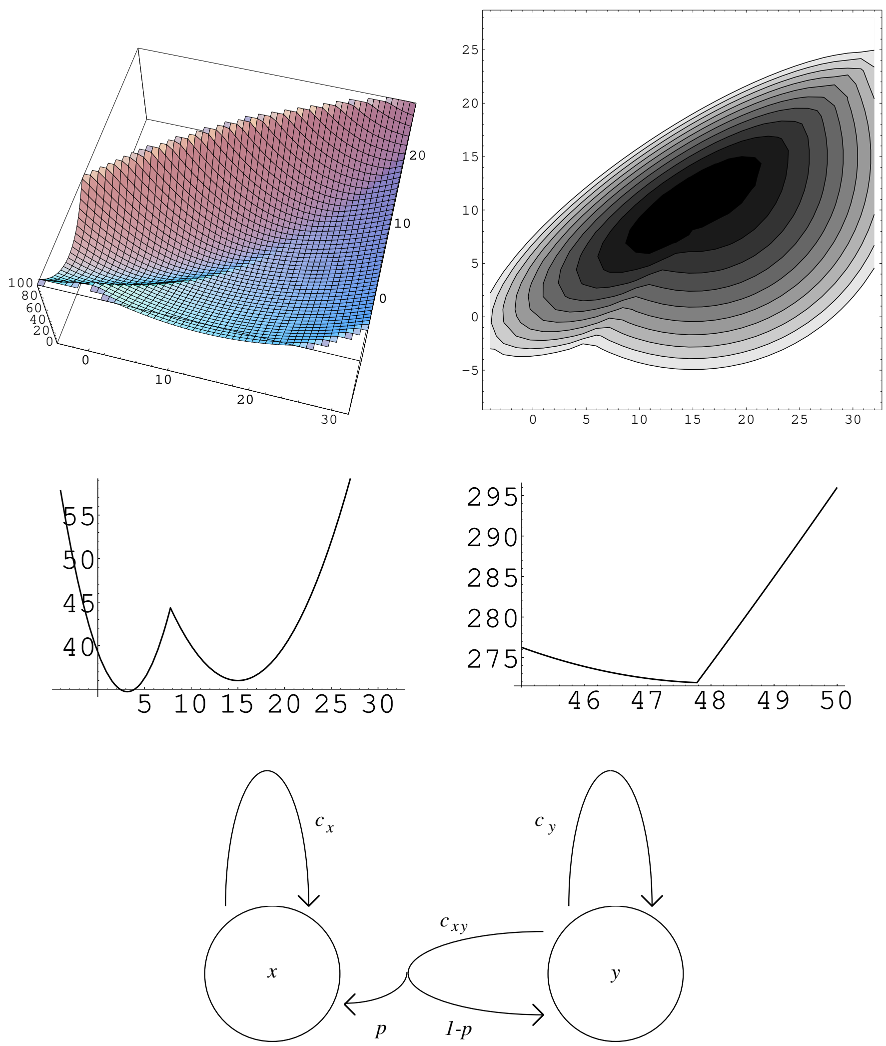


Figure 5.3: Several views of a Bellman error surface.

dimensional slices correspond to different one-parameter families of approximate representations for the value function. As the plots show, it is easily possible to have multiple local minima or derivative discontinuities at the minimum.

It may be possible to minimize Bellman error efficiently by using hybrid algorithms, for example damped Newton methods, Levenberg-Marquardt, or gradient descent with momentum. Baird has proposed a promising hybrid method which interpolates between temporal differencing (described below) and gradient descent [Bai95].

Even if we can find the parameters which minimize squared Bellman error, though, there is another important difficulty: not all Bellman errors are equally important. In some MDPs, many optimal paths pass through one or a small number of bottleneck states. Errors at the bottlenecks are more important than errors elsewhere: at the bottlenecks, a single error can affect many paths. If we simply minimize Bellman error, we may end up accepting an important error at a bottleneck instead of a larger but less important error at some other state. Worse, we can't sidestep the problem simply by weighting errors at the bottleneck states more heavily, since different policies can have different bottlenecks and we won't know which states are the real bottlenecks until we have already found the optimal policy.

There are heuristic algorithms which attempt to reweight states during the optimization procedure, but so far no such algorithm has been proven to converge for general function approximators. These algorithms can perform quite well in practice.

5.3.7 Multi-step methods

The Bellman constraint that corresponds to a transition from state x to state y with cost c is

$$v(x) \leq \gamma v(y) + c$$

This constraint relates the value of state x to the value of its immediate successor y. Similarly, the constraint that corresponds to a transition from y to z with cost d is $v(y) \leq \gamma v(z) + d$. Combining these two constraints gives

$$v(x) \leq \gamma^2 v(z) + \gamma d + c \tag{5.7}$$

Equation 5.7 relates the value of state x to the value of its two-step successor z.

We can combine three successive one-step constraints to make a three-step constraint, four to make a four-step constraint, and so forth. In an absorbing MDP, we can go so far as to combine all the transitions in an entire trajectory to make a single constraint of the form $v(x) \leq$ constant. Such an inequality is called a TD(1) constraint, by analogy to the TD(λ) algorithm described in Section 5.4.1. The advantage of a TD(1) constraint is that it is not recursive: it constrains the value of only one state rather than two. That means that we can use supervised learning algorithms to find approximate solutions to problems that contain only TD(1) constraints.

There may be many more multi-step constraints than there are one-step ones: if our MDP has a constant number of actions from each state, then (ignoring possible duplicates) the number of k-step constraints on $v(x)$ is exponential in k. (A degenerate case of this rule applies to Markov processes. For a Markov process the base of the exponential is 1, meaning that there is exactly one k-step constraint on $v(x)$ for each positive k.) To avoid dealing with an exponential number of constraints, many practical methods restrict their attention to multi-step constraints for transition sequences that actually occur in the observed data. For example, such methods would ignore the constraint (5.7) unless the learner had at some point moved from state x to state y to state z.

Multi-step constraints are redundant if we plan to solve the Bellman equations exactly. But, approximate methods for solving the Bellman equations may treat a multi-step constraint differently from its component one-step constraints. For example, for a Markov process we can define a k-step version of Equation 5.3 that looks like

$$A^{\mathrm{T}} \operatorname{Diag}(f) E^k A w = \dots$$

It is reasonable to ask whether approximate methods are likely to be more accurate if they use one-step or multi-step constraints. As a rough rule, one-step constraints are more data-efficient, while multi-step constraints are better at minimizing the effects of the function approximator. There is experimental evidence [Sut88] which suggests that a combination of constraints at different time scales works better than either single-step constraints or TD(1) constraints alone.

5.3.8 Stopping problems

Stopping problems are the subset of MDPs in which the agent has exactly two actions at each state: one action is called "continue" and has an arbitrary effect, and the other is called "stop" and leads immediately to the ending state $\odot$. The paper [TV97] points out that, unlike for general MDPs, there are still well-defined state visitation frequencies in a stopping problem: these are just the frequencies with which we would visit the nonterminal states if we never chose the stop action. So, it makes sense to solve the nonlinear equations

$$A^{\mathrm{T}} D \min(PAw + c, d) = A^{\mathrm{T}} D A w$$

where P is the transition probability matrix for continuing, c is the cost vector for continuing, d is the cost vector for stopping, D is the diagonal matrix whose entries are the state visitation frequencies, A is a matrix whose columns are basis vectors for representing the value function, and the minimum operation is taken componentwise. This expression is the analog of Equation 5.3. While the minimum operation makes the equations nonlinear, [TV97] gives a convergent algorithm for finding the solution.

5.3.9 Approximate policy iteration

It is possible to combine policy iteration with approximate methods for finding value functions. There are no such combinations that have been proven to converge for general MDPs and function approximators, but some combinations seem to work in practice. For example, the experiments in Chapter 4 use one such algorithm, and another is described in [BI96].

5.3.10 Policies without values

It is possible to learn a policy directly, without representing value functions along the way. For example, we can pick a starting policy, evaluate it by following some trajectories and measuring the incurred cost, and try to modify it to make it better. Methods for doing so include gradient descent, simulated annealing, and genetic algorithms.

Unlike simulated annealing and genetic algorithms, gradient descent requires the ability to compute an unbiased estimate of the gradient of a parameterized policy's expected cost with respect to one of its parameters. It is not obvious that it is possible to compute this gradient without reference to the value function, but [Wil92] gives an algorithm called Reinforce which does so.

The advantage of methods of this kind is that they try directly to optimize actual costs, instead of some proxy for actual costs like the consistency of a value function. The disadvantage of these methods is that they can be slow to converge: without the intermediate representation of a value function, it is harder to decide which parts of a policy are responsible for high costs.

Baird and Moore [BM99] have recently derived an algorithm called VAPS (for value and policy search) that can combine gradient descent on expected total cost with gradient descent on squared Bellman error or on other related performance measures. Such an algorithm can use a value function to decide which parts of a policy need modifying, but can also take actual costs into account directly.

5.3.11 Linear-quadratic-Gaussian approximations

It is common practice to approximate a nonlinear control problem by an LQG problem in some neighborhood. Unfortunately, a single linear-quadratic model is often not sufficient, and it is much harder to build a piecewise-LQG approximation to a control problem. The difficulty is in ensuring consistency along the edges of the pieces: the value function in each piece no longer satisfies the Ricatti equations, since it depends also on the values in every other piece.

One approach to this problem is to ignore it. That is, we can compute several separate LQG approximations around different points, ignoring possible interactions. Then we can control the system using the LQG approximation which is most appropriate for the current operating conditions, or by interpolating among several nearby models. This approach is called gain scheduling. It is particularly effective when the reward function is globally quadratic, as it is for example when we are trying to track a reference signal as closely as possible. In this case the LQG models can't get confused about where the lowest costs are, but only about how to get there. In addition, if the controller does get stuck far from the small costs, it is often possible to unstick it by hallucinating a series of target points (represented as a series of fictitious quadratic cost functions) which are close enough together that the linear-quadratic approximations can follow them and which lead the controller to a desirable region of state space. Of course, the question of which target points to use can be as difficult as the original control problem.

For control problems too difficult for gain scheduling, Atkeson has developed a method for growing "spines" backward along optimal trajectories [Atk94]. A spine comprises a series of local LQG models; each model is locally approximately consistent with the previous and subsequent models on the same spine, but models on different spines do not interact, so there are not too many dependencies between models.

Control methods based on linearization suffer from some problems. The first is that they may require a large number of linear pieces, forcing us either to store many precomputed controllers or to search for and generate controllers as needed in real time. The second and more important is that the system may not be even locally approximable by an LQG model: transition functions aren't always smooth, errors aren't always small and Gaussian, and arbitrarily large control inputs aren't always practical.

5.4 Incremental algorithms

Two of the best-known algorithms for finding value functions are TD(λ) and Q-learning [Sut88, Wat89]. Both of these algorithms are incremental, meaning that they examine each training example once and then forget it. This property may be useful if storage space is at a premium or if it is as easy to generate a new training example as it is to remember an old one. Q-learning solves Markov decision processes but does not handle function approximation, while TD(λ) can handle function approximation but only solves Markov processes.

5.4.1 TD(λ)

TD(λ) is an algorithm that finds approximate value functions for Markov processes. (TD is short for temporal differences, because the update for TD(λ) depends on the difference between parameters of successive states.) It can use any representation for value functions that is linear in its coefficients; that is, it can represent $v = Aw$ for any matrix A whose columns we want to use as basis vectors.

If we are given a Markov process, it is possible to discover the bottleneck states by observing actual or simulated trajectories from the process. (This is not true for an MDP, since the bottleneck states depend on the optimal policy.) By observing trajectories, we can build unbiased estimates of how often we visit each state. Once we know the state visitation frequencies, we can solve Equation 5.3 to find an approximate value function.

The TD(0) algorithm is an incremental algorithm which implicitly discovers the state visitation frequencies and solves Equation 5.3. After observing a transition from state i to state j at cost c, TD(0) updates its parameter vector w by the rule

$$w \leftarrow w + \eta a_i((\gamma a_j - a_i) \cdot w + c)$$

where η is a learning rate and a_i and a_j are the ith and jth rows of A expressed as column vectors. It is possible to show [Sut88, Day92, TV96] that under appropriate conditions TD(0) converges to the solution of Equation 5.3.

TD(λ) is a slightly more complicated algorithm with an update that depends on a whole sequence of states instead of just the last two. As the papers cited above show, it converges to the solution of an equation similar to Equation 5.3.

There is no straightforward way to generalize TD(λ) to solve Markov decision processes. Still, there are several popular heuristic MDP algorithms based on the method of temporal differences. These include TD-based variants of value iteration, Q-learning, policy iteration, and modified policy iteration. Perhaps the most successful is TD value iteration, which has surfaced for example in a world-class backgammon player [Tes94] and an elevator controller [CB96].

TD-based methods have the advantage that, at least heuristically, one would expect them to be good at finding bottleneck states because they always reweight each state based on how often the agent encounters it while following the current policy. Unfortunately, this advantage is only heuristic: no one has yet found a characterization of when these methods even converge, much less a proof that they end up with reasonable weights. In fact, it is possible to construct examples [Gor96, Ber96] where some of these methods oscillate forever between two or more policies with different value functions.

TD-based methods depend on being able to find out the state-visitation frequencies for each policy. (In fact, it is easy to cause them to diverge by visiting states at the wrong frequencies.) This fact is both an advantage and a disadvantage: while it allows TD-based algorithms to take bottleneck states into account easily and naturally, it means that all known implementations are based on following trajectories in either the real MDP or a model of it, which can be an efficiency disadvantage compared to non-TD-based algorithms.

5.4.2 Q-learning

It is difficult to write an incremental algorithm which directly learns the value function of a Markov decision process. The problem is the location of the nonlinearity in the Bellman equations: if we write

$$v^{(t+1)}(x) = \min_a \mathbb{E}\left[c(x,a) + \gamma v^{(t)}(\delta(x,a))\right]$$

then it is easy to get an unbiased estimate of the expectation for a single value of a, but it is hard to get an unbiased estimate of the minimum over all a.

To see why, imagine taking the minimum of two numbers, each corrupted by zero-mean random noise. The minimum will be below the true minimum if the noise in either number is negative, while it will be above the true minimum only if both numbers have positive noise [TS93].

To solve this problem, we can (as suggested in [Wat89]) break the Bellman equation into two pieces:

$$Q(x,a) = \mathbb{E}[c(x,a) + \gamma v(\delta(x,a))]$$

$$v(x) = \min_a Q(x,a)$$

If we write

$$Q^{(t+1)}(x,a) = \mathbb{E}\left[c(x,a) + \gamma \min_b Q^{(t)}(\delta(x,a), b)\right]$$

then it is easy to get an unbiased estimate of the expectation: we can sample c from the distribution of $c(x,a)$ and y from the distribution of $\delta(x,a)$ and compute $c + \gamma \min_b Q^{(t)}(y,b)$.

The Q-learning algorithm stores Q instead of v. On each step it samples a transition (say from state x to state y under action a at cost c) and updates

$$Q(x,a) \leftarrow (1-\eta)Q(x,a) + \eta(c + \gamma \min_b Q^{(t)}(y,b))$$

Under appropriate assumptions, [JJS94, Tsi94] prove that Q-learning converges with probability 1 to the true Q function.

5.5 Other methods

There is a long history of research into Markov decision processes and related problems, and we have only summarized a fraction of it here. Some interesting approaches not mentioned above are:

- Methods which assume a particular form of representation for the solution to the HJB equation, including [DS96] and [Goh93].

- Adaptive control (see, *e.g.*, [SL91]), which attempts to control a system containing unknown parameters by adapting parameter estimates online. The adaptation law may be chosen to try to reproduce the observed dynamics as accurately as possible (self-tuning control); or, more directly, it may try to reduce the tracking error between the observed trajectory and the trajectory predicted by an ideal reference model (model-reference adaptive control). General convergence guarantees usually require the model to have some special form, for example linear separately in the control inputs and the unknown parameters. Adaptive control techniques may be combined with the robust sliding mode control design described above. See [CS95] for a modern example of an adaptive control algorithm.

- Various neural-net approaches based on "unfolding" a problem by making a copy of the adjustable parameters for each time step. After unfolding, all variable dependencies are feedforward, so derivative calculations are simplified.

5.6 Summary

The research in this thesis extends the state of the art in several ways. To understand how, we can define the following hierarchy of function fitters. Each type of function approximation algorithm in the list includes and generalizes the previous ones.

Exact A degenerate case. Represents a function by storing its value at every possible input.

Piecewise constant Includes grids and other state aggregation.

Averager As defined in Chapter 2. Includes k-nearest-neighbor and linear and multilinear interpolation.

Linear Linear regression with an arbitrary basis, including for example polynomials, sines and cosines, and wavelets.

Generalized linear A linear function with a monotone transfer function applied to the output. Includes for example logistic regression.

General Everything else. Examples include neural nets and hierarchical mixtures of experts.

Before this thesis, the state of the art in learning value functions for general MDPs included algorithms that are guaranteed to converge when using exact or piecewise constant representations, or when using a limited subset of averagers. It also included algorithms that use general representations and can work well in practice, but are not guaranteed to converge. And, it included algorithms that can't handle fully-general MDPs but which can guarantee convergence with more-general representations than averagers, such as TD(λ) for Markov processes, or analytic solution of the HJB equations for some continuous control problems. Finally, the state of the art in worst-case learning included performance bounds for some generalized linear functions but not all.

Chapter 2 of this thesis advances the state of the art by defining an algorithm with guaranteed convergence that can represent value functions with arbitrary averagers. Chapter 3 advances the state of the art by extending worst-case regret bounds to cover a larger fraction of generalized linear function approximators. Finally, Chapter 4 takes the first steps towards an algorithm that can use arbitrary linear function approximators to represent value functions.

During the course of this thesis, other researchers have (of course) also advanced the state of the art in finding value functions. Of note are [TV94], which duplicated some of the results in Chapter 2; [SJJ95], which described an online algorithm related to fitted value iteration; [TV97], which extended TD(λ) to handle stopping problems; [MM98], which described a kind of averager that converges to the exact value function (in the limit of increasing representational power) when approximating a continuous-time MDP; and [Bai95] and [BM99], which developed gradient-descent style algorithms that are guaranteed to converge at least to a local maximum when using (differentiable) general representations.

Chapter 6

SUMMARY OF CONTRIBUTIONS

Finding approximate value functions for Markov decision processes is important because it addresses a basic need in machine learning: the need for the learner to find reasonable sequences of actions despite complicated, probabilistic environments. This thesis has presented three threads of research all motivated by the goal of approximating value functions.

The contributions of the research on fitted value iteration are to discover a class of function approximators that is compatible with fitted value iteration; to derive convergence and error bounds for fitted value iteration using approximators in this class; to reduce fitted value iteration to exact value iteration on an embedded process; and to perform experiments demonstrating that fitted value iteration is capable of solving Markov decision processes that require complex pattern recognition.

The contributions of the research on worst-case learning are to provide a framework in which to prove regret bounds for a wide variety of learning algorithms and to apply this framework to bring together known regret bounds and prove new ones. While we have not proven any bounds specifically about the problem of solving Markov decision processes, we expect that the results of this research will be helpful in proving such bounds, because the information available to a learner about an MDP is often not in the form of a sample of independent identically distributed random variables.

The contributions of the research on solving Markov decision processes by convex programming are to explore the connection among MDPs, convex optimization, and statistical estimation; to propose a new way to design algorithms for approximating value functions; and to experiment with new algorithms built according to this design. While the new algorithms do not improve on the best existing methods for approximating value functions, they do demonstrate that the design holds the promise of avoiding some of the shortcomings of current value function approximation methods.

These three threads of research work together to advance the state of the art in finding approximate solutions to Markov decision processes. Together they provide a wide variety of new tools for designing algorithms that allow learners to act appropriately in complicated, uncertain environments.

Bibliography

[AGMX96] Erling D. Andersen, Jacek Gondzio, Cszaba Mészáros, and Xiaojie Xu. Implementation of interior point methods for large scale linear programming. Technical Report 1996.3, University of Geneva, 1996.

[AHW96] Peter Auer, Mark Herbster, and Manfred Warmuth. Exponentially many local minima for single neurons. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1996.

[Atk94] C. G. Atkeson. Using local trajectory optimizers to speed up global optimization in dynamic programming. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan Kaufmann, 1994.

[Bai95] L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning (proceedings of the twelfth international conference)*, San Francisco, CA, 1995. Morgan Kaufmann.

[BB96] Steven J. Bradtke and Andrew G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57, 1996.

[BD59] R. Bellman and S. Dreyfus. Functional approximations and dynamic programming. *Mathematical Tables and Aids to Computation*, 13:247–251, 1959.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[Bel61] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[Ber76] Dimitri P. Bertsekas. *Dymanic Programming and Stochastic Control*. Academic Press, 1976.

[Ber95] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Massachusetts, 1995.

[Ber96] D. P. Bertsekas. Talk, 1996. Given at the NSF workshop on reinforcement learning.

[BI96] Dimitri Bertsekas and Sergey Ioffe. Temporal differences-based policy iteration and applications in neuro-dynamic programming. Technical Report LIDS-P-2349, MIT, 1996.

[BKK63] R. Bellman, R. Kalaba, and B. Kotkin. Polynomial approximation — a new computational technique in dynamic programming: allocation processes. *Mathematics of Computation*, 17:155–161, 1963.

[Bla65] D. Blackwell. Discounted dynamic programming. *Annals of Mathematical Statistics*, 36:226–235, 1965.

[BM95] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: safely approximating the value function. In G. Tesauro and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 7. Morgan Kaufmann, 1995.

[BM99] Leemon Baird and Andrew Moore. Gradient descent for general reinforcement learning. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11. MIT Press, 1999.

[BN78] Ole Barndorff-Nielsen. *Information and exponential families in statistical theory*. Wiley, New York, 1978.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[CB96] R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1996.

[CBFH$^+$95] Nicoló Cesa-Bianchi, Yoav Freund, David P. Helmboldt, David Haussler, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. Technical Report UCSC-CRL-95-19, University of California Santa Cruz, 1995.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[CS95] M. Cannon and J.-J. E. Slotine. Space-frequency localized basis function networks for nonlinear system estimation and control. *Neurocomputing*, 9(3), 1995.

[CT89] C.-S. Chow and J. N. Tsitsiklis. An optimal multigrid algorithm for discrete-time stochastic control. Technical Report P-135, Center for Intelligent Control Systems, 1989.

[Dav96] S. Davies. Multidimensional triangulation and interpolation for reinforcement learning. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1996.

[Day92] P. Dayan. The convergence of TD(λ) for general lambda. *Machine Learning*, 8(3–4):341–362, 1992.

[DS96] P. Dayan and S. P. Singh. Improving policies without measuring merits. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1996.

[DY79] Persi Diaconis and Donald Ylvisaker. Conjugate priors for exponential families. *Annals of Statistics*, 7(2):269–281, 1979.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[Fre96] Yoav Freund. Predicting a binary sequence almost as well as the optimal biased coin. In *Proc. 9th Ann. Workshop on Computational Learning Theory*, pages 89–98. ACM Press, 1996.

[GO77] D. Gottlieb and S. A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. SIAM, Philadelphia, 1977.

[Goh93] C. J. Goh. On the nonlinear optimal regulator problem. *Automatica*, 29(3):751–756, 1993.

[Gor95a] G. J. Gordon. Stable function approximation in dynamic programming. Technical Report CS-95-103, CMU, 1995.

[Gor95b] G. J. Gordon. Stable function approximation in dynamic programming. In *Machine Learning (proceedings of the twelfth international conference)*, San Francisco, CA, 1995. Morgan Kaufmann.

[Gor96] G. J. Gordon. Chattering in SARSA(λ). Internal report, 1996. CMU Learning Lab.

[Gor99] Geoffrey J. Gordon. Regret bounds for prediction problems. In *Proc. 12th Ann. Workshop on Computational Learning Theory*. ACM Press, 1999.

[HKW98] David Haussler, Jyrki Kivinen, and Manfred Warmuth. Sequential prediction of individual sequences under general loss functions. *IEEE Transactions on Information Theory*, 1998. To appear.

[JJS94] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, 1994.

[KW96] Jyrki Kivinen and Manfred K. Warmuth. Relative loss bounds for multidimensional regression problems. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1996.

[KW97] Jyrki Kivinen and Manfred K. Warmuth. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132(1):1–63, 1997. Preliminary version appeared as tech report UCSC-CRL-94-16; extended abstract appeared in 27th STOC.

[LW92] Nick Littlestone and Manfred Warmuth. The weighted majority algorithm. Technical Report UCSC-CRL-91-28, University of California Santa Cruz, 1992.

[MM98] Rémi Munos and Andrew Moore. Barycentric interpolators for continuous space & time reinforcement learning. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 10, Cambridge, MA, 1998. MIT Press.

[MN83] P. McCullagh and J. A. Nelder. *Generalized Linear Models*. Chapman & Hall, London, 2nd edition, 1983.

[Moo94] A. W. Moore. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan Kaufmann, 1994.

[OR70] James Ortega and W. C. Rheinboldt. *Iterative solution of nonlinear equations in several variables*. Academic Press, New York, 1970.

[Roc70] R. Tyrell Rockafellar. *Convex Analysis*. Princeton University Press, New Jersey, 1970.

[Ros83] Sheldon Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, 1983.

[Sab93] P. Sabes. Approximating Q-values with basis function representations. In *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum.

[Sam59] A. L. Samuels. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[SJJ95] S. P. Singh, T. Jaakkola, and M. I. Jordan. Reinforcement learning with soft state aggregation. In G. Tesauro and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 7. Morgan Kaufmann, 1995.

[SL91] J.-J. E. Slotine and W. Li. *Applied Nonlinear Control*. Prentice-Hall, New Jersey, 1991.

[SO91] A. Stuart and J. K. Ord. *Kendall's Advanced Theory of Statistics*. Oxford University Press, New York, 5th edition, 1991.

[Sut88] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[SY94] S. P. Singh and R. C. Yee. Technical note: an upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16(3):227–233, 1994.

[Tes90] G. Tesauro. Neurogammon: a neural network backgammon program. In *IJCNN Proceedings III*, pages 33–39, 1990.

[Tes94] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[TS93] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum.

[Tsi94] J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16(3):185–202, 1994.

[TV94] J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large-scale dynamic programming. Technical Report P-2277, Laboratory for Information and Decision Systems, 1994.

[TV96] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. Technical Report LIDS-P-2322, MIT, 1996.

[TV97] J. N. Tsitsiklis and B. Van Roy. Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing financial derivatives. Technical Report LIDS-P-2389, MIT, 1997. To appear in *IEEE Transactions on Automatic Control*.

[TZ93] Michael A. Trick and Stanley E. Zin. A linear programming approach to solving stochastic dynamic programs. Unpublished manuscript, 1993.

[TZ95] Michael A. Trick and Stanley E. Zin. Spline approximations to value functions: a linear programming approach. Unpublished manuscript, 1995.

[TZ97] Michael A. Trick and Stanley E. Zin. Spline approximations to value functions: A linear programming approach. *Macroeconomic Dynamics*, 1:255–277, 1997.

[Van94] Robert Vanderbei. LOQO: An interior point code for quadratic programming. Technical Report SOR 94-15, Princeton, 1994. Revised 11/30/98.

[Vov90] Volodimir Vovk. Aggregating strategies. In *Proc. 3rd Ann. Workshop on Computational Learning Theory*, pages 371–383. Morgan Kaufmann, 1990.

[Wat89] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, England, 1989.

[Wil92] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229, 1992.

[YSS97] Y. Yang, E. D. Sontag, and H. J. Sussman. Global stabilization of linear discrete-time systems with bounded feedback. *Systems and Control Letters*, 1997. To appear.