Reinforcement Learning and Mistake Bounded Algorithms

Yishay Mansour *

March 1, 1999

Abstract

Markov Decision Process (MDP) and Partially Observable MDP (POMDP) have become the model of choice in reinforcement learning. This work explores an interesting connection between mistake bounded learning algorithms and computing a near-best strategy, from a restricted class of strategies, for a given POMDP.

We show that if a class of strategies has a mistake bound algorithm that makes at most d mistakes, then there is an algorithm to compute a near-best strategy from the class in time polynomial in $1/\epsilon$, the accuracy parameter, $\log(1/\delta)$, the confidence parameter, H, the horizon parameter, and exponential in d, the mistake bound. Our transformation assumes only the ability to execute actions in the POMDP and the ability to reset the POMDP to its initial state.

1 Introduction

Reinforcement learning addresses the following ambitious goal: How can an agent, wondering in a potentially unknown environment, maximize its long-term return. The model underline most of the research in reinforcement learning is that of Markov Decision Process (MDP) [SB98, BT96]. An MDP includes a set of states, that describe the environment, and a set of actions, that the agent can perform in each state. When the agent performs an action, the MDP changes to a new state, in a probabilistic way, and the agent receives a probabilistic immediate reward and observes the new state of the MDP. The goal of the agent is to maximize a function of all the immediate rewards it receives, which is called the return. Widely studied return functions include: undiscounted bounded-horizon, discounted infinite-horizon, and undiscounted infinite-horizon.[1]

MDPs allows us to model both the uncertainty in the outcome of actions, in the immediate rewards and in the knowledge of the agent of the environment. An additional benefit is the rich literature that exists on MDPs in Operation Research and Control Theory [Ber95, Ber87]. This literature includes many efficient algorithms for MDPs. Specifically, for the above return functions, given a complete description of an MDP, we can compute an optimal policy[2] in polynomial time [How60, Bel57].

*AT&T Labs and Tel Aviv University. e-mail: `mansour@research.att.com`

[1]In undiscounted bounded-horizon, there is a parameter H, and the return is the sum of the first H immediate rewards. In discounted infinite-horizon, there is a parameter $\gamma \in (0,1)$, the immediate reward at the ith step is discounted by γ^i, and the return is the sum of the discounted immediate rewards. In undiscounted infinite-horizon the return is the average of the immediate rewards (in the limit).

[2]A policy is a mapping from states to actions. In an MDP the optimal action selection can be describe by a policy, i.e. it does not depend on the entire history but only on the current state.

Researchers in reinforcement learning have considered two important variations of MDP, which would make the model more appealing in practice. The first is Partially Observable MDP (POMDP), which is an MDP in which the agent does not observe the current state of the environment, but only some probabilistic function of it. (In many applications the entire state of the environment might be very complex, but the agent observes only a small relevant fraction of it.) Introducing POMDP makes many of the computational problems extremely hard. First, in order to maximize the return the agent needs a strategy, which selects an action depending on the entire history. Second, computing an optimal strategy is PSPACE-complete for undiscounted finite-horizon [PT87], and EXPTIME-complete for undiscounted infinite-horizon [Lit96]. The second extension is to consider exponential size state space. The classical algorithms for MDPs run in time polynomial in the state space, however in many cases we would like to have an exponential size state space, e.g. each state is an n-bit vector. In such a case even giving a complete description of the model would be prohibitively large.

The above two issues, as well as others, motivate the restriction of the strategy to come from a restricted class of strategies. As in supervised learning, we would like the strategy class to have two conflicting properties. On the one hand we hope that it is expressive enough, so that it includes strategies with high return. On the other hand, we choose a simple strategy class so that finding a good strategy is feasible. Restricted strategy classes are widely used in practice, for example, in the backgammon playing program the policy is implemented using a neural network [Tes94].

The problem of finding a near-best strategy from a given class of strategies resembles the classical question in supervised learning of finding a near-best hypothesis from a given class of hypotheses. However there are crucial differences that distinguish the two settings. In supervised learning we assume that there is some target function, that labels the examples, and some *fixed* distribution that generates the examples. A crucial property is that the distribution is fixed for all the hypotheses. This implies that the same sample can be evaluated on any hypothesis, and (hopefully) the observed error estimates well the true error

In reinforcement learning the case is very different. There is no fixed distribution, rather each strategy defines a *different* distribution over trajectories[3]. By choosing a strategy we define both a hypothesis and a distribution. This added degree of freedom, which is inherent in reinforcement learning, causes the problem to be much more complex. If we use one strategy to generate a sample of trajectories, it is not clear how to use this sample to estimate the return of another strategy. The problem is that the other strategy might generate a very different distribution, and in the extreme case the support of the two distributions might be disjoint.

From a theoretical perspective there are two issues here. The first is information theoretic. How much experience do we need to generate, and in what way to generate it, so that we have enough information to determine the near-best strategy. This is the equivalent of the sample size issue in supervised learning (with the added complication that we need to decide *how* to generate the experience). The second is a computational issue. Given that information theoretically we have enough information, how can we computationally, efficiently, find a near-best strategy. This is the equivalent to the computational issue of the supervised learning problem, finding a good hypothesis efficiently.

A recent work [KMN99] addresses the information theoretic issue of generating enough information to determine the near-best strategy with respect to a finite-horizon return. Using a random policy (i.e. each time select a random action), they generate a set of finite horizon trajectories. Given this set of trajectories they define for each strategy in the class an estimate and show that the estimates uniformly converge to the

[3]A trajectory is a complete description of an execution in a POMDP.

true return. The number of trajectories they generates grows exponentially in H, the horizon parameter, and polynomial in ϵ, the accuracy parameter, V_{max}, the maximum possible return, $\log(1/\delta)$, the confidence parameter, and the VC-dimension of the strategy class. (Note that the linear dependence on the VC-dimension implies, implicitly, that there is a very good reuse of trajectories.) The complexity issue, i.e., how to find efficiently a strategy with the highest estimated return, is not address in [KMN99].

In this work we establish a connection between mistake bounded algorithms and computing a near-best strategy from a given class with respect to a finite-horizon return (define in Section 2). We assume that we receive, as a black box, an algorithm that learns the strategy class with at most d mistakes. Using this algorithm we generate trajectories in the POMDP, following various strategies in the class, and determine a near-best strategy from the strategy class. Our algorithm runs in time polynomial in H, ϵ, V_{max}, and $\log(1/\delta)$, and exponentially in d.

Our result improves on [KMN99] in two important way. First, we address the computational issue. Assuming that the mistake bounded algorithm is efficient, then our algorithm is efficient. Second, the dependence on H, the horizon parameter. In this work the number of trajectories (and time) grows like H^d in contrast to 2^H in [KMN99], which is a significant saving for $d \ll H$.

We also show a lower bound for the special case where the strategies depend only on the last observed value. Our lower bound states that for any such strategy class $\mathcal{F}$, that has mistake bound $d \leq H$, there is an MDP $M_{\mathcal{F}}$, such that any algorithm that estimates in $M_{\mathcal{F}}$ a near-best policy for $\mathcal{F}$ has to generate a number of trajectories that grows exponentially with d and polynomially with $1/\epsilon$, and V_{max}.

The paper is organized as follows. In Section 2 we define the various learning models. Section 3 gives an algorithm for the special case of $d = 1$. This case is somewhat simpler, yet it has most of the ingredients of the general case, which is described in Section 4. The lower bound is sketch in Section 5.

2 Preliminaries

2.1 Partially Observable Markov Decision Processes

A *Markov Decision Process* (MDP) is define by a tuple (S, A, s_0, P, R), where S is a set of states, A is a set of actions, s_0 is the initial state, P is the next-state distribution, and R is the immediate reward function. The *next-state distribution* P defines for each state-action pair (s, a), a next-state distribution $P(s'|s, a)$ that specifies the probability of transition to each state s' upon execution of action a from state s. (Note that $\sum_{s'} P(s'|s, a) = 1$.) The *immediate reward function* R, for each state-action pair (s, a), defines a random variable $R(s, a)$. We assume that the immediate rewards are bounded, i.e., $|R(s, a)| \leq R_{max}$. An agent wondering in an MDP has access to the current state s, the agent selects an action to execute (say action a), then the agent observes an immediate reward r (distributed according to $R(s, a)$) and the next state s' (distributed according to $P(s'|s, a)$).

A *Partially Observable Markov Decision Process (POMDP)* consists of an underlying MDP and *observable distributions* $Q(o|s)$ for each state s, where o is a random variable called the *observable value* at state s. An agent wondering in a POMDP has access only to the observable values rather then the underline states. (Since different states may have the same observable value, the knowledge of the agent in a POMDP is significantly limited compared to the MDP case.) The walk of the agent is described by a trajectory, which is a sequence $t = (o_0 a_0 r_0 \ldots o_i a_i r_i \ldots)$, where o_i is the observable value at ith state, a_i is the action the agent executed at the ith state, and r_i is the immediate reward it received. The function $\texttt{prefix}(t,i)$ returns the prefix of t up to the observable value of the ith state, i.e. $(o_0 a_0 r_0 \ldots o_{i-1} a_{i-1} r_{i-1} o_i)$. The function $\texttt{action}(t,i)$ returns a_i.

In our abstraction, when the agent is wondering it can either execute an action in the POMDP or reset the POMDP to its initial state. Executing an action a in the POMDP M is done by the function $\texttt{POMDP-action}(M,a)$. If M is in state s, then after $\texttt{POMDP-action}(M,a)$ it moves to state s' (distributed according to $P(s'|s,a)$). The function $\texttt{POMDP-action}(M,a)$ returns a pair (o,r) where o is the observable value at s' (distributed according to $Q(o|s')$) and r is the immediate reward (distributed according to $R(s,a)$). A reset is done by executing the function $\texttt{POMDP-reset}(M)$, which resets the state of M to the initial state s_0. The function $\texttt{POMDP-reset}(M)$ returns a value o, which is the observable value at s_0 (distributed according to $Q(o|s_0)$). Each $\texttt{POMDP-reset}(M)$ terminates the current trajectory and starts a new trajectory. We assume that each function invocation takes one time unit.

A trajectory includes many immediate rewards. We like to combine this sequence of immediate rewards, $(r_0, \cdots r_i \cdots)$, to a single *return* $\mathcal{R}$, and let the goal of the agent be to maximize the return. In this paper we focus on returns which may be computed (or well approximated) using the first H steps, where H is a parameter, and are bounded in absolute value by V_{max}. This includes two well studied return functions. The first is the *undiscounted finite-horizon* return, which is $\mathcal{R} = \sum_{i=1}^{H} r_i$, where H is the finite-horizon parameter. In this case, by definition, $\mathcal{R}$ depends only on the first H steps, and $V_{max} = HR_{max}$. The second is the *discounted infinite-horizon* return. Here there is a discount factor $\gamma \in (0,1)$, and the return is $\mathcal{R} = \sum_{i=0}^{\infty} \gamma^i r_i$. In this case we can approximate $\mathcal{R}$ using the first $H_\epsilon = \log_\gamma \epsilon / V_{max}$ immediate rewards, i.e. $\mathcal{R}' = \sum_{i=0}^{H_\epsilon} \gamma^i r_i$, and $V_{max} = R_{max}/(1-\gamma)$ is the maximum return (since each immediate reward r_i is bounded in absolute value by R_{max}). This implies that $\mathcal{R} - \mathcal{R}' < \epsilon$, and hence using H_ϵ steps we can approximate $\mathcal{R}$ within ϵ. Also note that $H_\epsilon < (1/(1-\gamma)) \log(V_{max}/\epsilon)$.

We do not concern ourselves with a specific return function, and rather define an abstract function $\texttt{RETURN}(t)$, whose value is the return of trajectory t, and a function $\texttt{RETURN-H}(t)$, which estimates the return of t using only the first H immediate rewards in t. We assume that $H = H_\epsilon$ is such that $|\texttt{RETURN}(t) - \texttt{RETURN-H}(t)| \leq \epsilon$, and hence the added error by considering only the first H_ϵ steps is bounded by ϵ.

The function by which the agent selects the actions it executes is called a *strategy*. In general, the agent's strategy at the ith step may depend on the entire history up to it, i.e., $(o_0 a_0 r_0 \ldots o_{i-1} a_{i-1} r_{i-1} o_i)$. Therefore the input to a strategy is a trajectory while the output is an action (or a distribution over actions). The random variable T_f describes a random trajectory t generated by following strategy f, where the randomness is over the next-state distribution, the immediate reward function and any randomization of f.

Our main interest is to find near-best strategies from a restricted class of strategies $\mathcal{F}$. Given a POMDP M, we define $opt(M,\mathcal{F})$ to be $\sup_{f \in \mathcal{F}} E_{T_f}[\texttt{RETURN}(T_f)]$. We call a strategy f an ϵ-near-best strategy for $\mathcal{F}$ if,

$$opt(M,\mathcal{F}) - E[\texttt{RETURN}(T_f)] \leq \epsilon.$$

We distinguish between two cases, according to whether f belongs to $\mathcal{F}$. The *realizable* case is when we restrict the ϵ-near-best strategy to be a member of $\mathcal{F}$, while in the *un-realizable* case the ϵ-near-best strategy can be arbitrary (still we are comparing its expected return to $opt(M,\mathcal{F})$).

In order to make the presentation simpler, we assume that there are only two actions, i.e. $A = \{0,1\}$. However, it is straightforward to generalize the results to any finite number of actions. Our second limitation is concern with strategies. In general a strategy may be stochastic, but in this specific work we will limit our interest to deterministic strategies. This is due to the connection we establish with mistake bounded algorithm, which are almost always deterministic (since they minimize the worst case number of mistakes).

2.2 Mistake bounded algorithms

The Mistake Bounded Model [Lit88] is using an adversarial online model to learn a given class of functions $\mathcal{F}$. An online algorithm A works in iterations. In each iteration A is given some input x, A replies with a value b, and then A is given the correct label c. A mistake occurs if $b \neq c$. The aim of the algorithm A is to minimize the number of mistakes it makes, given that the inputs are labeled using some function $f \in \mathcal{F}$. An algorithm A is a mistake bounded algorithm for function class $\mathcal{F}$ if there exists an a priori bound d, such that for any sequence of inputs the algorithm A does not make more than d mistakes. There are no assumption about how the input sequence is chosen.

We view the functions in $\mathcal{F}$ in two different ways. On the one hand they are Boolean functions, mapping inputs to $\{0, 1\}$, while on the other hand they are strategies, mapping trajectories to actions $\{0, 1\}$. A mistake bounded algorithm A for $\mathcal{F}$ receives inputs, which are trajectories, and output a response, which we will interpret as an action. However, this semantics would be transparent to the algorithm A. Furthermore, A does not expects to see its inputs in any particular order.

When we describe an algorithm formally, we assume the following interface to the algorithm A. First, the algorithm A is initialized using `initialize`(A). The algorithm A is given input x using the function `input`(A, x) that returns a value b, which is the reply of A. Later, the procedure `correct`(A, x, c) informs A that the correct value of x is c. We assume that each function invocation takes one time unit.

We say that a mistake bounded algorithm A for learning $\mathcal{F}$ is *realizable* if at any point the function it defines is in $\mathcal{F}$. More precisely, during any iteration we can view the output of `input`$(A, \cdot)$ as a function $g(\cdot)$. The algorithm A is realizable if it is always the case that $g \in \mathcal{F}$. (In case that the mistake bounded algorithm A is realizable, then the strategies that our algorithm will generate would be also from the class $\mathcal{F}$, thus we are in the realizable case of computing an ϵ-near-best strategy for $\mathcal{F}$.)

2.3 VC dimension and Chernoff bounds

We start by defining the VC-dimension for a class of Boolean functions. Let $\mathcal{F}$ be a set of Boolean functions defined over domain $\mathcal{X}$. We say that $\mathcal{F}$ *shatters* a set $S = \{x_1, \ldots, x_d\} \subset \mathcal{X}$ if for each binary vector $b_1 \cdots b_d \in \{0, 1\}^d$, there exists a function $f \in \mathcal{F}$ such that $f(x_i) = b_i$. The *Vapnik-Chervonenkis (VC) dimension* of $\mathcal{F}$ is the maximum size of a set S that $\mathcal{F}$ shatters. (If there are sets of arbitrary size, then the VC dimension is infinite.) The following theorem states the ability of the VC dimension to characterize the sample complexity (see, [KV94, Vap82]).

Theorem 2.1 *Let $\mathcal{F}$ be a class of Boolean function with VC dimension d, and D an arbitrary distribution over $\mathcal{X}$. Given $f_t \in \mathcal{F}$, define $f \in \mathcal{F}$ as ϵ-bad if $\mathbf{Pr}_D[f(x) \neq f_t(x)] \geq \epsilon$. Let S be a sample of size $m(\epsilon, \delta, d)$ drawn i.i.d from D. Then with probability $1 - \delta$, for every ϵ-bad function $f \in \mathcal{F}$, there is an*

$x \in S$ such that $f(x) \neq f_t(x)$, given that,

$$m(\epsilon, \delta, d) \geq c\left(\frac{d}{\epsilon}\log\frac{1}{\epsilon} + \frac{1}{\epsilon}\log\frac{1}{\delta}\right),$$

for some constant $c > 0$.

The following is a Chernoff bound for the convergence of binary random variables (see, e.g. [AS92]).

Lemma 2.2 *Let $X_1, \ldots, X_n$ i.i.d. random variables, with $|X_i| \leq 1$ and $E[X_i] = p$. For $n \geq (1/\epsilon^2)\log(1/\delta)$, with probability $1-\delta$, we have that,*

$$|\frac{1}{n}\sum_{i=1}^{n} X_i - p| \leq \epsilon.$$

3 A single mistake algorithm

In this section we consider the simple case where we have a strategy class $\mathcal{F}$ and an algorithm A, such that for any $f \in \mathcal{F}$ and any input sequence, the algorithm A makes at most one mistake. (A simple example of such a class, is a set of strategies, g_o, one for each observable value, and $g_\emptyset$. The strategy $g_o(t)$ performs action 1 if the last observable is o and performs 0 otherwise. The strategy $g_\emptyset$ always performs action 0.) Our aim is to use algorithm A to compute an ϵ-near-best strategy from $\mathcal{F}$.

The idea of the algorithm is as follows. We first fix the initial function of A denoted by f_0. The function $f_0(x)$ represents the response of A on input x (as the first input). (We can compute $f_0(x)$ by initializing A, giving it input x, and the response of A is $f_0(x)$.) We use f_0 as our initial strategy to generate ℓ_0 trajectories of length H starting at s_0. Namely, at each step we compute f_0 on the trajectory that we observed so far. The output of f_0 is the next action we execute.

Using the ℓ_0 trajectories it is rather straightforward to approximate the return of strategy f_0 (by simply averaging the return of the ℓ_0 trajectories). Our main claim would be that this estimate is also a good estimate for many of the other strategies. Specifically, if a strategy f has a negligible probability of generating a trajectory (of length H starting at s_0) different from that of f_0, then the return of the two strategies has to be very similar.

Given a trajectory $t = (o_0 a_0 r_0 \ldots o_H a_H r_H)$, we say that f is consistent with t, if for every i, we have $a_i = f(t_i)$, where $t_i = \texttt{prefix}(t, i)$. Otherwise f is not consistent with the trajectory t. (We also say that a trajectory t is consistent with f, which has the same meaning.) Let $e(f_1, f_2)$ be the probability that a trajectory t (of length H starting at s_0) following strategy f_1, is not consistent with f_2. (Note that the measure is symmetric, since the strategies are deterministic, therefore trajectories that are consistent are given the same probability under both strategies.) The following lemma captures our claim that if $e(f_1, f_2)$ is small, then the return of the two strategies is similar.

Lemma 3.1 *For any two strategies f_1 and f_2,*

$$|E[\texttt{RETURN}(T_{f_1})] - E[\texttt{RETURN}(T_{f_2})]| \leq e(f_1, f_2) V_{\max}.$$

Proof: Let V_0 be the expected reward on the trajectories that both f_1 and f_2 are consistent with, and V_i is the expected reward of f_i on the remaining trajectories. We can write

$$E[\mathtt{RETURN}(T_{f_i})] = (1 - e(f_1, f_2))V_0 + e(f_1, f_2)V_i.$$

Therefore,

$$|E[\mathtt{RETURN}(T_{f_1})] - E[\mathtt{RETURN}(T_{f_2})]| = e(f_1, f_2)|V_1 - V_2|.$$

Since by definition $|V_i|$ is bounded by V_{max} the lemma follows. □

We showed that if $e(f_0, f)$ is small then the returns of f and f_0 are similar. Hence, a good estimate for f_0 is also a good estimate for f. In the next lemma we show that with high probability, any strategy f such that $e(f_0, f)$ is large, then f is not consistent with at least one of the ℓ_0 trajectories generated following f_0. We prove here a general lemma which we later apply also for the general case.

Lemma 3.2 *Let $\mathcal{F}$ be a class of strategies that can be learned in the mistake bounded model using at most d mistakes and $f_0 \in \mathcal{F}$. Let S be a set of $m(\epsilon/H, \delta, d)$ trajectories of length H starting from s_0 and following f_0. With probability at least $1 - \delta$, for each $f \in \mathcal{F}$, such that $e(f_0, f) \geq \epsilon$, there is a trajectory $t \in S$, such that f is not consistent with t.*

Proof: Let $e_i(f)$ be the probability that f is not consistent with a trajectory generated by f_0 due to step i, and is consistent up to that point. Clearly, $e(f_0, f) = \sum_{i=1}^{H} e_i(f)$. If $e(f_0, f) \geq \epsilon$ we have that for some i, $e_i(f) \geq \epsilon/H$. Let D_i be the distribution of the trajectories generated by f_0, up to step i. The prefixes $t_i = \mathtt{prefix}\ (t, i)$ of the trajectories $t \in S$ are an i.i.d sample from D_i.

Since $\mathcal{F}$ has a mistake bounded algorithm with a at most d mistakes, it implies that its VC dimension is at most d [Lit88]. By Theorem 2.1, if we take a sample S of size $m(\epsilon/H, \delta, d)$ from D_i, then with probability at most δ there exists an $f \in \mathcal{F}$, such that $e_i(f) \geq \epsilon/H$ and f is consistent with all the t_is. Therefore, with probability at least $1 - \delta$, any function $f \in \mathcal{F}$ for which $e(f_0, f) \geq \epsilon$, there is a trajectory $t \in S$, such that f is not consistent with t. □

The second stage of the algorithm uses the sample to identify certain strategies it estimates directly. For each prefix t_i of a trajectory $t \in S$ we generate a strategy f_{t_i}. The function $f_{t_i}(x)$ is defined using A as follows: (1) initialize A, (2) give A input t_i, (3) inform A that it made a mistake on t_i, (4) give A the input x, (5) the reply of A on x is $f_{t_i}(x)$.

The number of strategies we define in the second stage is at most $H\ell_0$. For each such strategy, f_{t_i}, we generate ℓ_1 trajectories following f_{t_i}, and estimate f_{t_i} return by the average return of the ℓ_1 trajectories. By Lemma 2.2, for $\ell_1 \geq (1/\epsilon^2)\log(H\ell_0/\delta)$ we are guaranteed that with probability $1 - \delta$ for each of those strategies the approximation error is at most ϵ.

We now define for each strategy $f \in \mathcal{F}$ an estimate denoted by $Estimate[f]$. For the strategy f_0 the value of $Estimate[f_0]$ is the average return of the ℓ_0 trajectories. For each strategy f_i, that is consider in the second stage, the value of $Estimate[f_i]$ is the average return of the ℓ_1 trajectories generated following f_i. For any other strategy $f \in \mathcal{F}$ we assign $Estimate[f] = Estimate[f_0]$. (A detailed description of the algorithm is found in Figure 1.)

Lemma 3.3 *Let $\ell_0 = \max\{m(\epsilon/(2V_{\max}), \delta/3, d), (1/\epsilon^2)\log(3/\delta)\}$ and $\ell_1 = (1/\epsilon^2)\log(3H\ell_0/\delta)$. With probability $1 - \delta$ for every $f \in \mathcal{F}$ we have*

$$|Estimate[f] - E[\mathtt{RETURN}(T_f)]| \leq \epsilon$$

Proof: By Lemma 3.2, with probability $1-\delta/3$, for any f such that $e(f, f_0) \geq \epsilon/(2V_{max})$, there is some $t \in S$ such that f is not consistent with t. Let t_i be the prefix of t on which f disagree with f_0, i.e., $f_0(t_i) \neq f(t_i)$. Since algorithm A learns the class $\mathcal{F}$ with at most one mistake, it implies that after A makes an mistake on t_i, it does not make any more mistakes. Therefore there is only one function $f \in \mathcal{F}$ for which $f_0(t_i) \neq f(t_i)$. (If there was another function $f' \in \mathcal{F}$ for which $f_0(t_i) \neq f'(t_i)$, then we can force A to make a second mistake.) In the second stage we approximate the return of each such f_{t_i} directly, using ℓ_1 trajectories. With probability $1-\delta/3$ all our estimates in the second stage have error at most ϵ.

Again, by Lemma 3.2, any strategy that is not evaluated directly in the second stage, with probability $1-\delta/3$ has $e(f, f_0) \leq \epsilon/(2V_{max})$. By Lemma 3.1 this implies that $|E[\texttt{RETURN}(T_f)] - E[\texttt{RETURN}(T_{f_0})]| \leq \epsilon/2$. By Lemma 2.2, with probability $1-\delta/3$, we have $|Estimate(f_0) - E[\texttt{RETURN}(T_{f_0})]| \leq \epsilon/2$. Since any strategy $f \in \mathcal{F}$ that is not approximated directly in the second stage, by definition $Estimate[f] = Estimate[f_0]$, we have that $|Estimate[f] - E[\texttt{RETURN}(T_f)]| \leq \epsilon$. □

We can now state our theorem for the case that A makes at most one mistake.

Theorem 3.4 *Let $\mathcal{F}$ be a class of strategies, and A is a realizable mistake bounded algorithm that makes at most one mistake. Then algorithm* `estimate-POMDP-strategy-one-mistake`*, with probability $1-\delta$ finds an ϵ-near-best strategy and runs in time $O(H^2\ell_0\ell_1) = poly(H, V_{\max}/\epsilon, \log(1/\delta))$.*

4 General mistake bounded algorithms

In this section we extend the algorithm of the previous section to the general case, where the strategy class $\mathcal{F}$ can be learned by a mistake bounded algorithm A with at most d mistakes.

The main idea is to design a recursive algorithm. The high level idea is the following. Given a function class $\mathcal{F}$, we identify a subclass $\mathcal{F}_0 \subset \mathcal{F}$, for which we are able to immediately approximate the return (without any recursive call). We also identify at most L subclasses $\mathcal{F}_i \subset F$, such that $\mathcal{F}_i$ can be learn using at most $d-1$ mistakes. Finally, for completeness, we need to require that each function $f \in \mathcal{F}$ belongs to some subclass, i.e. $F = \cup_i \mathcal{F}_i$. We recursively compute for each $\mathcal{F}_i$, $i > 0$, an ϵ-near-best strategy f_i and return the strategy f_i and an estimated v_i of its return. In addition we have a strategy f_0 that is ϵ-near-best strategy for $\mathcal{F}_0$ with an estimated return v_0. Given all the values of the recursive calls, the ϵ-near-best strategy for $\mathcal{F}$ is the strategy f_i with the maximum v_i.

The time complexity is bounded as follows. Since at each step we create at most $L+1$ subclasses, and make at most L recursive calls, then in the overall process we do not make more then $(L+1)^d$ recursive calls. We now have to show three things. First, how we define $\mathcal{F}_0$ and the approximates the return of its strategies. Second, how we create at most L subclasses each with mistake bound at most $d-1$. Third, that the union of the subclasses covers all the strategies in $\mathcal{F}$.

The idea is very similar to the single mistake algorithm. We start again with the basic strategy of A, namely f_0. We use f_0 to generate a set S of ℓ trajectories. The subclass $\mathcal{F}_0$ includes all the strategies which are consistent with all the trajectories in S. For each strategy in $\mathcal{F}_0$ we estimate its return by the estimated return of f_0, which we denote by v_0. By Lemma 3.1 we are guarantee that this is a good estimate for any strategy for which $e(f_0, f)$ is small. By Lemma 3.2 we are guarantee that if $e(f_0, f)$ is large then, with high probability, f is not consistent, i.e., $f \notin \mathcal{F}_0$.

The trajectories in S are used to define the subsets $\mathcal{F}_i$. For each trajectory $t \in S_0$ we consider every prefix t_i, and define a subclass $\mathcal{F}_{t_i}$ which includes all the strategies $f \in \mathcal{F}$ for which $f_0(t_i) \neq f(t)$. Clearly there is an algorithm that make at most $d-1$ mistakes on $\mathcal{F}_{t_i}$. (Namely, the algorithm A after making a mistake on t_i.) Now we can recursively compute the ϵ-near-best strategy for each $\mathcal{F}_i$. Assume that the ϵ-near-best strategy for $\mathcal{F}_i$ is f_i and its estimated return is v_i. Our estimate of the ϵ-near-best strategy for $\mathcal{F}$ is the strategy f_i that maximizes v_i. (See Figure 2 for a full description of the algorithm `estimate-POMDP-strategy`.)

For our analysis we add another variable which estimates the return of every $f \in \mathcal{F}$, denoted by $Estimate[f, \mathcal{F}]$. When we compute the estimated value for $\mathcal{F}$ we assume that we have the estimated values for each $f \in \mathcal{F}_i$, $i > 0$, i.e. $Estimate[f, \mathcal{F}_i]$. If $f \in \mathcal{F}_0$ then $Estimate[f, \mathcal{F}] = Estimate[f_0, \mathcal{F}]$. Otherwise, let i be the least index such that $f \in \mathcal{F}_i$. We assign $Estimate[f, \mathcal{F}] = Estimate[f, \mathcal{F}_i]$.

Lemma 4.1 *Let $\ell = \max\{m(\epsilon/(2V_{\max}), \delta, d), (1/\epsilon^2)\log(1\delta)\}$. Let $\mathcal{F}$ be a strategy class for which algorithm A makes at most d mistakes. With probability $1-(\ell H+1)^d\delta$ for every $f \in \mathcal{F}$ we have,*

$$|Estimate[f, \mathcal{F}] - E[\texttt{RETURN}(T_f)]| \leq \epsilon.$$

Proof: The proof is by induction on d. The proof for $d=0$ is immediate, since in such a case $\mathcal{F}$ includes a single function. Assume that the lemma holds for $d-1$ and we show that it holds for d.

In the first stage we consider the strategy f_0. Using it we sample a set S of ℓ trajectories. Given the sample S let $\mathcal{F}_0$ be all the strategies that are consistent with all the trajectories in S.

By Lemma 3.2, with probability $1-\delta$, any strategy $f \in \mathcal{F}$, for which $e(f_0, f) \geq \epsilon/(2V_{max})$, has a trajectory $t \in S$, such that f is not consistent with some prefix t_i of t. (In such a case $f \in \mathcal{F}_{t_i}$.) Therefore, with probability $1-\delta$, all the strategies $f \in \mathcal{F}_0$ have $e(f_0, f) < \epsilon/(2V_{max})$. This implies, by Lemma 3.1, that every $f \in \mathcal{F}_0$ can be well approximated by the strategy f_0. Since we have an approximation for f_0 such that, with probability $1-\delta$, $|Estimate[f_0, \mathcal{F}] - E[\texttt{RETURN}(T_{f_0})]| \leq \epsilon/2$, it implies that $|Estimate[f, \mathcal{F}] - E[\texttt{RETURN}(T_f)]| \leq \epsilon$, for every $f \in \mathcal{F}_0$.

By the definition of $\mathcal{F}_0$, for each $f \in \mathcal{F} - \mathcal{F}_0$ we have some trajectory $t \in S$, such that $f \in \mathcal{F}_{t_i}$, where t_i is some prefix of t. We define a mistake bounded algorithm A_{t_i} that learns $\mathcal{F}_i$ with at most $d-1$ mistakes as follows. The algorithm A_{t_i} uses algorithm A in the following way. When initializing A_{t_i}, it does the following: (1) initializes A, (2) gives A input t_i, (3) gets A response b, and (4) responds that A made a mistake. Later when A_{t_i} is given an input x, it passes x to A, and replies A's reply.

Now we can use the inductive hypothesis for each $\mathcal{F}_{t_i}$. By the inductive hypothesis for each $f \in \mathcal{F}_{t_i}$, with probability of at least $1-(\ell H+1)^{d-1}\delta$,

$$|Estimate[f, \mathcal{F}_{t_i}] - E[\texttt{RETURN}(T_f)]| \leq \epsilon.$$

Since we have at most ℓH subclasses $\mathcal{F}_{t_i}$, plus the subclass $\mathcal{F}_0$, our success probability is at least $1-(\ell H+1)^d\delta$. $\square$

We need to choose the value of ℓ such that the probability of failure would be bounded by δ. Let ℓ be the minimal integer that satisfies $\ell > \max\{m(\epsilon/(2V_{max}), \delta/(\ell H+1)^d, d), (1/\epsilon^2)\log(1/\delta)\}$. We have,

$$\ell = O(\frac{d}{\epsilon}\log\frac{dH}{\epsilon} + \frac{1}{\epsilon^2}\log\frac{1}{\delta}\}) = poly(H, V_{max}/\epsilon, \log(1/\delta), d).$$

The following theorem is immediate from the Lemma 4.1, using the above value of ℓ.

Theorem 4.2 *Let $\mathcal{F}$ be a class of strategies, and A is a realizable mistake bounded algorithm that makes at most d mistakes on $\mathcal{F}$. With probability $1-\delta$ the algorithm* `estimate-POMDP-strategy` *computes an ϵ-near-best strategy for $\mathcal{F}$ in time $O(H^2\ell(H\ell+1)^d)$.*

5 Lower Bounds

In this section we sketch the lower bound on the amount of exploration required to compute an ϵ-near-best strategy. Our lower bound applies only to strategies that depend only on the last observable value.

We use the characterization of the mistake bound for a given class given by [Lit88]. Let $\mathcal{F}$ be a class of Boolean functions. We define a binary tree labeled from $\mathcal{F}$ as follows. Each internal node is labeled by an input, and each edge is labeled by either one or zero. Each leaf contains some function $f \in \mathcal{F}$ that is consistent with the path from the root to the leaf. The rank of $\mathcal{F}$ is the maximum rank of any binary tree[4] labeled by $\mathcal{F}$. Littlestone [Lit88] showed that the rank of the class $\mathcal{F}$ is identical to its mistake bound. The following claim is a straightforward implication of this.

Claim 5.1 *Let $\mathcal{F}$ be a function class with mistake bound d, then there exists a complete binary tree of depth d labeled from $\mathcal{F}$.*

Given a class $\mathcal{F}$ with mistake bound d, by Claim 5.1 we have a complete binary tree of depth d labeled by $\mathcal{F}$. We use this tree to construct POMDP $M_{\mathcal{F}} = (S, A, s_0, P, R, Q)$. The states of $M_{\mathcal{F}}$ are the nodes of the tree plus a special "terminate" state. The actions $A = \{0, 1\}$. The initial state s_0 is the root of the tree. We define the transition function P as follows. If in the tree s_b is the b-child of s, where $b \in \{0, 1\}$, then performing action b in state s moves the POMDP to state s_b, i.e. $P(s_b|s, b) = 1$. From each leaf with any action we move to a special "terminate" state, which we cannot exit. The observable values are define as follows. Each internal node, sm in the tree, is labeled by some u. In $M_{\mathcal{F}}$ when we are in state s we observe value u, i.e. $Q(u|s) = 1$. (Note that the transition function and the observable values are deterministic.)

We define the immediate reward function R of $M_{\mathcal{F}}$ as follows. We choose a random leaf v. The immediate reward for executing any action in v is V_{max} with probability $1/2+\epsilon$ and 0 with probability $1/2-\epsilon$. For any other leaf v' and any action a the immediate reward is V_{max} with probability $1/2$ and 0 with probability $1/2$. For any other state and action the immediate reward is 0.

Given $M_{\mathcal{F}}$ each deterministic strategy defines a path from the root to a leaf (and then to the terminate state). By the construction of the tree, there is at least one strategy in $\mathcal{F}$ that reaches each leaf. We choose the immediate reward function so that only strategies reaching the leaf v would be ϵ-near-best.

Assume that we give the learner all the functions except for the immediate reward function. Then his task is reduced to the following task: given 2^d coins, one of which is biased, determine the biased coin (with probability $1-\delta$) and with the least expected number of coin flips. A similar setting, for the multi-arm bandit problem was studied in [ACBFS95], and using their lower bound we prove the following lower bound in our setting in our setting.

Theorem 5.2 *Let $\mathcal{F}$ be a strategy class such that any $f \in \mathcal{F}$ depends only on the last observable value. Assume that any algorithm A that learns $\mathcal{F}$ in the mistake bounded model makes at least d mistakes.*

[4] see [CLR90] for the definition of a rank of a tree.

There exists an POMDP $M_{\mathcal{F}}$, such that any algorithm, that finds an ϵ-near-best policy for $\mathcal{F}$ in $M_{\mathcal{F}}$, with probability at least 1/2, generates at least $\Omega(2^d (V_{\max}/\epsilon)^2)$ trajectories.

References

[ACBFS95] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *36th Annual Symposium on Foundations of Computer Science*, pages 322–331, 1995.

[AS92] N. Alon and J. Spencer. *The Probabilistic Method.* Wiley, 1992.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, N.J., 1957.

[Ber87] Dimitri P. Bertsekas. *Dynamic Programming: deterministic and stochastic methods.* Prentice-Hall, 1987.

[Ber95] Dimitri P. Bertsekas. *Dynamic Programming and optimal control.* Athena Scientific, 1995.

[BT96] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic programing.* Athena Scientific, 1996. Deals with various techniques related to Neural networks.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[How60] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press, 1960.

[KMN99] M. Kearns, Y. Mansour, and A. Ng. Approximate planning in large pomdps via reusable trajectories. In *ICML*, 1999. submitted.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *Computational Learning Theory.* MIT press, 1994.

[Lit88] Nick Littlestone. Learning when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[Lit96] Michael L. Littman. *Algorithms for sequential decision making.* PhD thesis, Brown University, 1996.

[PT87] Christos H. Papadimitrio and John N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operation Research*, 12(3):441–450, 1987.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning.* MIT press, 1998.

[Tes94] Gerald Tesauro. TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, 6:215–219, 1994.

[Vap82] V. N. Vapnik. *Estimation of Dependences Based on Empirical Data.* Springer-Verlag, New York, 1982.

1. Generate ℓ_0 trajectories, $t_1^0, \ldots, t_{\ell_0}^0$, of length H starting from s_0 following strategy f_0. Let r_i^0 be the return of trajectory t_i^0, i.e. $r_i^0 = \texttt{RETURN}(t_i^0)$.

2. The estimated return of f_0 is the average return of the ℓ_0 trajectories, i.e., $Estimate[f_0] = (1/\ell_0)\sum_{i=1}^{\ell_0} r_i^0$.

3. Let $S_0 = \{v | v = \texttt{prefix}(t_i^0, j)\}$.

4. For each $v \in S_0$ define $f_v(x)$ as follows. On input x, the function $f_v(x)$ returns the reply of algorithm A to input x, in the second step, after that in the first step A made a mistake on input v.

5. For each f_v, $v \in S$, generate ℓ_1 trajectories, $t_1^v, \ldots, t_{\ell_1}^v$, of length H starting from s_0 following strategy f_v. Let r_i^v be the return of trajectory t_i^v, i.e. $r_i^v = \texttt{RETURN}(t_i^v)$.

6. The estimated return of f_v is the average return of the ℓ_1 trajectories, i.e., $Estimate[f_v] = (1/\ell_1)\sum_{i=1}^{\ell_1} r_i^v$.

7. Let $F_1 = \{f_v | v \in S\}$. For any $f \in F - F_1$, we estimate its return to be the same as f_0, i.e., $Estimate[f] = Estimate[f_0]$.

8. Return the strategy for which the estimate is the highest, i.e., let $f_{max} = \texttt{argmax}_{f \in F} Estimate[f]$.

Figure 1: Algorithm `estimate-POMDP-strategy-one-mistake`

```
/* The function approx receives a bound on the number of mistakes, d, and a restriction U =
((t_1, b_1), ···, (t_k, b_k)) on the class of strategies F and returns a near best strategy from F_U. (F_U includes
all the strategies in F which are consistent with U.) */

FUNCTION  approx(d:integer, U:restrictions) returns (max: real, max-f :function)
   max ← −1;
   FOR i = 1 TO  ℓ DO                   /* generate ℓ trajectories */
   t ← generate(U);                     /* generate a trajectory following f_U */
   IFd > 0 THEN                  /* check for d = 0 */
      FOR j = 1 TO H DO                  /* perform a recursive call for each prefix */
          t_i ← prefix(t, i);                  /* consider the history until the ith action */
          b_i ← action(t, i);                  /* extract the ith action */
          (tmpmax, f) ← approx(d − 1, U||(t_i, b̄_i));                  /* recursive call with a different action */
          update-max(tmpmax, f, max, max-f);                  /* update the best strategy */
          END-FOR
      END-IF
   update-max(RETURN-H(t), U, max, max-f);                  /* compare best strategy to f_U */
   END-FOR
   return (max,max-f).

FUNCTION value(U: restriction, x:input) returns b;                  /* defines the function f_U */
initialize(A);                  /* initialize algorithm A */
FOR i = 1 TO |U| DO;                  /* simulate the sequence U with A.*/
     a ← input(A, t_i);                  /* Input the i-th input of U to A*/
     correct(A, t_i, b_i);                  /* update A by the desired value of t_i */
     END-FOR
b ← input(A, x) ;                  /* query A on input x*/
return b;                  /* return A response to x */

FUNCTION generate(U: restriction) returns (t: trajectory);          /* generate trajectory from M following f_U*/
o_0 ← POMDP-reset(M);                  /* reset the POMDP to initial state */
t ← o_0;                  /* initialize the trajectory to the first observable */
FOR i = 1 TO H DO;                  /* build a trajectory of length H */
     a ← value(U, t);                  /* compute next action */
     (o, r) ← PODMP-action(M, a);                  /* perform next action */
     t ← t||a||r||o;                  /* extend the trajectory */
     END-FOR
return t;

PROCEDURE update-max(R_1:real, U_1:restriction, R_2:real, U_2:restriction)
IFR_1 ≥ R_2 THEN
   U_2 ← U_1; R_2 ← R_1;
return
```

Figure 2: This is the code for the estimate-POMDP-strategy algorithm. A restriction $U = ((t_1, b_1), \cdots, (t_k, b_k))$ is a sequence of pairs (t_i, b_i), where t_i is a trajectory and b_i is an action. We relate a restriction U to a function by simulating the online algorithm A on it (this is done in the function **value**). To compute an ϵ-near-best strategy from $\mathcal{F}$ we invoke **approx**$(d, \emptyset)$.