Learning to Take Actions

Roni Khardon
Aiken Computation Laboratory,
Harvard University,
Cambridge, MA 02138
roni@das.harvard.edu

Abstract

We formalize a model for supervised learning of action strategies in dynamic stochastic domains, and show that pac-learning results on Occam algorithms hold in this model as well. We then identify a particularly useful bias for action strategies based on production rule systems. We show that a subset of production rule systems, including rules in predicate calculus style, small hidden state, and unobserved support predicates, is properly learnable. The bias we introduce enables the learning algorithm to invent the recursive support predicates which are used in the action strategy, and to reconstruct the internal state of the strategy. It is also shown that hierarchical strategies are learnable if a helpful teacher is available, but that otherwise the problem is computationally hard.

Introduction

Planning and acting have been mainly studied in AI with a logical perspective, where knowledge about the world is encoded in declarative form. In order to achieve goals, one proves that they are true in some world state, and as a side effect derives a plan for these goals (McCarthy 1958). Similarly, in partial order planning declarative information is given, and search in plan space is performed to find a plan (Weld 1994). However, the computational problems involved in these approaches are computationally hard (Cook 1971; Bylander 1994). Furthermore, these approaches have difficulties in handling dynamic situations where "re-planning" is used, and situation where the world is non-deterministic, or partially observable.

A different approach is taken by the reinforcement learning paradigm where "reactive" action selection is used. In this model an agent wanders in a (partially observable) Markov decision process, and the only source of information is a (positive or negative) reinforcement signal given in response to its actions. The goal of the agent is to find a good mapping from situations to actions so as to maximize its future reinforcement. While interesting results on convergence to optimal strategies have been obtained (Sutton 1988; Fiechter 1994), the resulting strategies essentially enumerate the state space, and therefore require exponential space and time.

Explanation Based Learning (EBL) (DeJong & Mooney 1986; Mitchell, Keller, & Kedar-Cabelli 1986) uses declarative knowledge and search, but learns from its experience, essentially compiling its knowledge into a more procedural form by saving generalized forms of the results of search as rules in the system. While arbitrary addition of rules may actually reduce the performance, utility based tests for added rules were found to be useful (Minton 1990). Note that, similar to reinforcement learning, EBL is an unsupervised process since no external guidance for the search is given, and that both approaches ultimately try to find the optimal solution to problems.

In this paper we follow the framework of learning to reason (Khardon & Roth 1994; 1995) and previous formalizations of learning in deterministic domains (Tadepalli 1991; 1992; Tadepalli & Natarajan 1996) and suggest a new approach to these problems. The new formalization, *learning to act*, combines various aspects of previous approaches. In particular we use the stochastic partially observable world model as in reinforcement learning, but on the other hand use symbolic representations and action strategies that are similar to the ones used in planning and explanation based learning.

Our model is similar to the reinforcement learning approach, in that the agent tries to learn action strategies which are successful in the world; namely, no explicit reasoning power is required from the agent. Rather, it is sufficient that an agent chooses its actions so that most of the time it succeeds.

Our framework differs from previous approaches in a few aspects. First, *no direct assumptions* on the structure of the world are made. We do assume that the world behaves as a partially observable Markov process, but we do not make any restrictions on the size or structure of this process.

On the other hand, in order to ensure tractability we assume that some *simple* strategy provides good behavior in the world, where simple is properly quantified. We also assume some form of *supervised learning*, where the learner observes a teacher acting in the world, and is trying to find a strategy that achieves comparable performance.

Unlike previous models we *do not require optimal performance* (which in many cases is hard to achieve), but rather

demand that the learner be able to reproduce things that have already been discovered. This can be seen as an attempt to model progress in some communities, where most agents only perform local discoveries or learning. However, once an important tool is found and established, it is transferred to the rest of the community relatively fast, and with no requirement that everyone understand the process properly, or reinvent it.

Another important part of this work is the choice of knowledge representation. We concentrate on action strategies in the form of Production Rule Systems (PRS). This is a class of programs which has been widely studied (Anderson 1983; Laird, Rosenbloom, & Newell 1986). A nice property of PRS is that it allows for a combination of condition action rules, and declarative knowledge that can be used for search, under the same framework. Previous studies have mainly used PRS in a manner similar to EBL emphasizing the effect of declarative representations.

In this paper, the rules are mainly used as a functional representation, which chooses which actions to take. Learning of simple PRS strategies, is performed with the help of an external teacher. Our strategies have a flavor of reactive agents. However, they are goal based, have internal state, and use predicates which compute simple recursive functions of the input.

We start by presenting the model of acting and learning in the world, and deriving a general learning result showing the utility of Occam algorithms. We then present a particular subset of PRS which we show learnable using this result, and briefly discuss the learnability of hierarchical strategies. We conclude with a discussion and some reference to future work. For lack of space, some details and proofs and further discussion are omitted from the paper; these can be found in (Khardon 1995).

The Model

Technically, the framework presented here is similar to the one studied in (Tadepalli 1991; Tadepalli & Natarajan 1996). The main difference is that we do not incorporate assumptions about the deterministic structure of the world into the model. Intuitively, the world is modeled as a randomized state machine; in each step the agent takes an action, and the world changes its state depending on this action. The agent is trying to get to a state in which certain "goal" conditions hold.

The interface of the agent to the world is composed of three components:

- The measurements of the learner are represented by a set of n literals, $x_1, x_2, \ldots, x_n$, each taking a value in[1] $\{0,1\}$. The set $X = \{0,1\}^n$ is the domain of these measurements.

  For structural domains, similar to (Haussler 1989), the input, a multi-object scene, is composed of a list of objects and values of predicates instantiated with these objects.
- The agent can be assigned a goal, from a previously fixed set of goals $\mathcal{G}$. For simplicity we would assume that $\mathcal{G}$ is the class of conjunctions over the literals $g_1, g_2, \ldots, g_n$, and their negations, where g_i represents the desired state of x_i. (This is similar to conjunctive goals in STRIPS style planning problems.)
- The agent has at its disposal a set of actions $O = \{o_1, \ldots, o_n\}$. (The choice of n as the number of actions is simply intended to reduce the number of parameters used.) In the learning model, the agent is not given any information on the effects of the actions, or the preconditions for their application. In particular, there is no hidden assumption that the effects of the actions are deterministic, or that they can be exactly specified.

The protocol of acting in the world is modeled as a infinitely repeated game. At each round, nature chooses an instance, (x, g), such that $x \in X$ and $g \in \mathcal{G}$. Then the agent is given some time, say N steps (where N is some fixed polynomial in the complexity parameters), to achieve the goal g starting with state x. In order to do this the learner has to apply its actions, one at a time, until its measurements have a value y which satisfies g (i.e. $g(y) = 1$).

Intuitively, each action that is taken changes the state of the world, and at each time point the agent can take an action and then read the measurements after it. However, some of the actions may not be applicable in certain situations, so the state does not have to change when an action is taken. Furthermore, the state may change even when no action is taken.

Definition 1 (strategy) *A* strategy *s is composed of a state machine (I, i_0, δ_s), and a mapping $s : X \times \mathcal{G} \times I \rightarrow O$ from instances and states into actions. An agent is following a strategy s if before starting a run it is in state i_0, and whenever it is in state $i \in I$, and on input (x, g), the agent chooses the action $s(x, g, i)$, and changes its state to $\delta_s(x, g, i)$.*

Most of the strategies we consider are stationary, namely no internal state is used. In this case a strategy is simply a mapping $s : X \times \mathcal{G} \rightarrow O$ from instances into actions.

Definition 2 (run) *A* run *of a strategy s on instance (x, g), is a sequence resulting from repeated applications of the strategy s,*

$$R(s, x, g) = x, s(x, g, i_0), x^1, s(x^1, g, i_1), x^2, \ldots,$$

until g has been achieved or N steps have passed, where for each $j \geq 1$, $i_j = \delta_s(x^{j-1}, g, i_{j-1})$.

Definition 3 (successful run) *A run is* successful *if for some $i \leq N$, $g(x^i) = 1$.*

Notice that, depending on the world, a run might be a fixed value or a random variable.

Definition 4 (world) *The world W is modeled as a partially observable Markov decision process whose transitions are effected by the actions of the agent.*

Given this definition, for any fixed starting state, a probability distribution is induced on the values that the run may

[1] Our results also hold in a more general model, where a third value, $*$, is used, denoting that the value of some variable is not known or has not been observed (Valiant 1995).

take. It should be noted that we do not make any assumptions on the size or structure of W. Furthermore, in contrast with reinforcement learning, we do not expect an agent to have complete knowledge of W. Instead, an agent needs to have a strategy that copes with its task when interacting with W.

When interacting with the world the agent has some form of a reset button which draws a new problem to be solved. We assume that, at the beginning of a random run, a state of the Markov process is randomly chosen according to some fixed probability distribution D. This distribution induces a probability distribution D over the measurements $X \times \mathcal{G}$ that the learner observes at a start of a run.

Definition 5 (random run) *A* random run *of a strategy s with respect to a world W, and probability distribution D, denoted $R(s, D)$, is a run $R(s, x, g)$ where (x, g) are induced by a random draw of D, and the successor states are chosen according to the transition matrix of W.*

The above definition ensures that a random run is indeed a random variable. Finally,

Definition 6 (quality of a strategy) *The quality $Q(s, D)$ of a strategy s, with respect to a world W, and probability distribution D, is*

$$Q(s, D) = \texttt{Prob}[R(s, D) \textit{ is successful}]$$

where the probability is taken over the random variable R (which is determined by D and W).

We study a supervised learning scenario, where the learner can observe a teacher acting in the environment. We assume that a teacher has some strategy t according to which it chooses its actions.

Definition 7 *The oracle* `example`(t) *when accessed, returns a random sample of $R(t, D)$.*

A learning algorithm will get access to the oracle `example` and will try to find a strategy which is almost as good as the teacher's strategy.

Definition 8 (learning) *An algorithm A is a* **Learn to Act** *algorithm, with respect to a class of strategies S, class of worlds $\mathcal{W}$, and class of distributions $\mathcal{D}$, if there exists a polynomial $p()$, such that on input $0 < \epsilon, \delta < 1$, for all $t \in S$, for all $W \in \mathcal{W}$, and for all $D \in \mathcal{D}$, and when given access to* `example`(t)*, the algorithm A runs in time $p(n, 1/\epsilon, 1/\delta)$, where n is the number of attributes A observes, and with probability at least $1 - \delta$ outputs a strategy s such that $Q(t, D) - Q(s, D) \leq \epsilon$.*

Learning Action Strategies

We now present a general result on learning algorithms. Similar to results in the PAC model (Blumer *et al.* 1987), we show that in order to learn it is sufficient to find a concise action strategy which is consistent with the examples given by the teacher.

The main idea is that an action strategy which is very different from the teacher's strategy will be detected as different by a large enough random sample. Notice that the distribution of the states visited by the agent within a run depends on the actions it takes. Therefore the states visited are not independent random variables, and PAC results are not directly applicable. Nevertheless, the examples runs are independent of each other. The proof of the next theorem follows by showing that most of the good runs of the teacher are also covered by a consistent strategy.[2]

We say that a strategy is consistent with a run $R = x, o_{i_1}, x^1, o_{i_2}, x^2, o_{i_3}, \ldots, o_{i_l}, x^l$ if for all j, the action chosen by the strategy in step j, given the history on the first $j-1$ steps (which determine the internal state of the strategy) is equal to o_{i_j}.

Theorem 1 *Let H be a class of strategies, and let L be an algorithm such that for any $t \in H$, and on any set of runs $\{R(t, D)\}$, L finds a strategy $h \in H$ which is consistent with all the runs. Then L is a learn to act algorithm for H when given $m = \frac{1}{\epsilon} \log(\frac{|H|}{\delta})$ independent example runs.*

Using the above theorem we can immediately conclude that several learning results developed in the model with a deterministic world hold in our model as well. In particular macro tables (Tadepalli 1991), and action strategies which are intersection closed and have a priority encoding over actions (Tadepalli & Natarajan 1996) are learnable in our model.

Representation of Strategies

Production rule systems (Anderson 1983; Laird, Rosenbloom, & Newell 1986) are composed of a collection of condition action rules $C \rightarrow A$, where C is usually a conjunction, and A is used to denote an action. Actions in PRS denote either a real actuator of the agent, or a predicate which is "made true" if the rule is executed. PRS are simply a way to describe programs with a special kind of control mechanism. An important part of this mechanism is the *working memory*. The working memory captures the "current state" view of the system. Initially, the input is put into the working memory, and the PRS then works in iterations. In each iteration, the condition C of every rule is evaluated, to get a list of rules which may be executed. Out of these rules, one is selected, by the "resolution mechanism", and its action A is executed. That is either the actuator is operated, or the predicate mentioned as A is added to the working memory. The above cycle is repeated until the goal is achieved.

We study the learnability of a restricted form of PRS. In particular we use a priority list of rules as a resolution mechanism, and restrict the conditions to include conjunction of bounded size. Furthermore, we restrict the amount and size of the working memory used. The working memory includes internal predicates and small state machines, and is combined with a particular control structure.

Blocks World: In order to illustrate the style of PRS considered, we present a PRS for the Blocks World. We then proceed with formal definitions.

[2] It is straightforward to generalize the theorem so that the hypothesis size will depend on the size of the strategy being learned as in (Blumer *et al.* 1987).

A situation in the blocks world is described by listing the names of blocks, and the relations that hold for them. The input relations we consider are: $clear(x)$ which denotes that nothing is placed above block x, and $on(x, y)$ which denotes that block x is on block y. We assume that the goal situation is described in a similar manner using the predicate $G()$. For example $G(on(a, b))G(on(b, c))$ could be our goal. The only action available is $move(x, y)$ which moves object x to be on y given that both were $clear$ beforehand.

Finding an optimal solution for this problem is NP-complete, but there is a simple algorithm that produces at most twice the number of steps that is needed (Gupta & Nau 1991). The idea is that if a block is above another block, which is part of the goal but is not yet in its goal place, then it has to be moved. If we arbitrarily move such blocks to the table, then we can easily build the required tower by moving each block once more. So blocks are moved twice, and each of them must be moved at least once in the optimal solution. We present a PRS which implements this algorithm (which assumes for simplicity that the target towers start on the table).

Our PRS has three parts. The first part computes the *support predicates* of the system. The second part consists of a priority list of condition action rules which we will refer to as the *main part of the PRS*. The third part includes rules for updating the *internal state*.

The production rule system first computes the support predicates $inplace(y)$, and $above(x, y)$. These have the intuitive meaning; namely $inplace(x)$ if x is already in its goal situation, and $above(x, y)$ if x is in the stack of blocks which is above y.

1. $inplace(T)$
2. $on(x, y) \wedge G(on(x, y)) \wedge inplace(y) \rightarrow inplace(x)$
3. $on(x, y) \rightarrow above(x, y)$
4. $on(x, y) \wedge above(y, z) \rightarrow above(x, z)$

Then the actions are chosen:

1. $clear(x) \wedge clear(y) \wedge G(on(x, y)) \wedge inplace(y) \rightarrow move(x, y)$
2. $inplace(y) \wedge G(on(x, y)) \wedge \overline{on(x, y)} \wedge above(z, x) \wedge clear(z) \wedge \overline{sad} \rightarrow move(z, T)$
3. $inplace(y) \wedge G(on(x, y)) \wedge \overline{on(x, y)} \wedge above(z, y) \wedge clear(z) \rightarrow move(z, T)$
4. $inplace(y) \wedge G(on(x, y)) \wedge \overline{on(x, y)} \wedge above(z, x) \wedge clear(z) \rightarrow move(z, T)$

Then the internal state is updated:

1. $sad \leftarrow sad \oplus inplace(y) \wedge G(on(x, y)) \wedge \overline{on(x, y)} \wedge above(z, x) \wedge clear(z)$

The PRS computes its actions as follows: (1) First, the support predicate rules are operated until no more changes occur. (No ordering of the rules is required.) (2) Then, the main part of the PRS chooses the action. The main part of the PRS is considered as a priority list. Namely, the first rule that matches the situation is the one that chooses the action. (It is assumed that if the condition holds for more than one binding of the rule to the situation, then an arbitrary fixed ordering is used to choose between them.) (3) The internal state is updated after choosing the action. The form of the transition rules is defined by conditions under which the value of a state variable is flipped; the details are explained below. The state machine is not so useful in the strategy above. (The internal state sad is superfluous and rule 4 could replace rule 2.) However, it is useful for example when memory for a particular event is needed.

While the PRS we consider have all these properties we restrict these representations syntactically. First, for simplicity we would assume that n bounds the number of objects seen in a learning scenario, the number of predicates, and the number of action names.

Restricted Conjunctions: When using PRS, and when the input is given as a multi-object scene, one has to test whether the scene satisfies the condition of a rule under any substitution of objects for variables. This binding problem is NP-hard in general, and a simple solution (Haussler 1989) is to restrict the number of object variables in the conjunction by some constant. We restrict the rules, so that there are at most k literals in the conjunction, and that every literal has at most a constant $C = 2$ object variables. This bounds the number of variables in a rule to be $2k + 2$. (A similar restriction has been used in (Valiant 1985).)

Notice that the priority list only resolves between different rules in the list. We must specify which action to choose when more than one binding matches the condition of a rule. For this purpose we use a priority preference between bindings. In particular, for every condition-action rule we allow a separate ordering preference on its bindings. (This includes as a special case the situation where all rules use the lexicographical ordering as a preference ordering.)

Support Predicates: The support predicates include a restricted type of *recursive* predicates, similar to the ones in the example. In particular, we consider support predicates that can be described using condition action rules $C \rightarrow A$, where C has at most k literals. Another restriction we employ is that the support predicates do not interact with each other; namely, the PRS can use many support predicates, but each of them is defined in terms of the input predicates. We consider both recursive and non-recursive predicates. A non-recursive predicate $l(x, y)$ is defined by a rule of the form $C \rightarrow l(x, y)$.

For a recursive predicate, we allow the same predicate to appear positively in the conjunction. We further associate with this predicate, a "base case rule". This rule is a non-recursive statement as above. For example, for the predicate $above$ in our example, $on(x, y) \rightarrow above(x, y)$ is the base case, and $on(x, y) \wedge above(y, z) \rightarrow above(x, z)$ is the recursive rule. The total number of predicates in this setting is bounded by m_0^2, where $m_0 = n(n + 1)^k(2k + 2)^{(2k+2)}$.

Notice that recursive predicates enhance the computing power of PRS considerably. These predicates enable the PRS to perform computations which are otherwise impossible. For example the predicate $above$ cannot be described by a simple PRS. Evaluating this predicate may require an arbitrary number of steps, depending on the height of the

stack of blocks.

Internal State: We next describe the restrictions on the state machines, whose transition function is restricted in a syntactic manner. Suppose the machine has c Boolean state variables $s_1, s_2, \ldots, s_c$. The transition function of the machine is described for each variable separately. For each s_i we have a k-conjunction as before, conjuncted in turn with an arbitrary conjunction of the state variables. This conjunction identifies a condition under which the value of s_i is to be flipped. For example we can have $s_1 \leftarrow s_1 \oplus s_2\overline{s_3}on(x,z)move(x,y)$, meaning that if s_2 was 1, s_3 was 0, and a certain block was just moved in a certain way, then we should flip the value of s_1. We would assume that the state machine is reset to state 0^c in the beginning of every run. The number of state machines that can be defined in this way is at most $m_2 = m_0^{2kc}(2k+2)^{c(2k+2)}3^{c^2}$.

PRS and Decision Lists

To illustrate the basic idea consider propositional PRS with no internal state and with no support predicates. The PRS is composed only of its main part, which is a priority list of condition action rules. Each condition is a conjunction of at most k literals, and the actions specify a literal $o_i \in O$. The class of PRS is very similar to the class of decision lists. Rivest (1987) showed that a greedy algorithm succeeds in finding a consistent hypothesis for this class; we observe that the same holds in our model.

First, recall that by Theorem 1 it is sufficient to find a strategy consistent with the examples in order to learn to act. Since the strategies are stationary we can partition each run into situation-action pairs and find a PRS consistent with the collection of these pairs, just as in concept learning.

The main observation (Rivest 1987) is that the teacher's strategy t is a consistent action strategy. Suppose we found a rule which explains some subset of the situation-action pairs. Then, if we add t after this rule we get a consistent strategy. Therefore, explaining some examples never hurts. Furthermore, there is always a consistent rule which explains at least one example, since the rule in t does. By enumerating all rules and testing for consistency we can find such a rule, and by iterating on this procedure we can find a consistent PRS.

Learning PRS in Structural Domains

Assume first, that the strategies are stationary. Therefore it is sufficient to consider situation-action pairs, which we refer to as examples. Notice that for our PRS the main part is in the form of a priority list. This part of the PRS can be learned as in the propositional case by considering rules with structural predicates. (The restrictions make sure that the number of rules is polynomial for fixed k). When testing to see whether a rule $C \rightarrow A$ is consistent with the examples, one has to check whether there is a binding order which is consistent for this rule. If an example matches the rule with two bindings, and only one of them produces the correct action, then we record a constraint on the binding order for this rule. (The binding producing the correct action must precede the other one.) The rule is consistent with the examples iff the set of constraints produced in this way is not cyclic.

Algorithm Learn-PRS

Initialize the strategy S to the empty list.
For each possible state machine Do:
  Compute all possible support predicates for all examples.
  Separate the example runs into a set E of situation-action pairs.
  Repeat:
    Find a consistent rule $R = C \rightarrow A$.
    Remove from E the examples chosen by R.
    Add R at the end of the strategy S.
  Until $E = \emptyset$ or there are no consistent rules.
If $E = \emptyset$ then output S and stop.
Otherwise, initialize the strategy S to the empty list, and go to the next iteration.

Figure 1: The Algorithm *Learn-PRS*

Our learning algorithm must also find the internal parts, not observable to it, namely the support predicates and state machine. To find the support predicates, we pre-process the examples by computing, for each example, the values of *all* possible invented predicates that agree with the syntactic restriction. (As noted above the number of such predicates is polynomial.) This computation is clearly possible for the non-recursive predicates. For the recursive predicates, one can use an iterative procedure to compute these values. First apply the base case on all possible bindings. Then in each iteration apply the recursive rule until no more changes occur. This procedure is correct since the recursive rule is monotone in the new predicate.

To find the internal state machine we must deal with non-stationary strategies. This problem is much harder in general, since the actions selected can make the impression that the output of the teacher is random or arbitrary. However, if we know the values of the state variables then the problem is simple again, since the strategy is stationary when these values are added to the input. The restrictions imposed imply that the number of state machines is polynomial. Therefore we can enumerate the machines, and for each possibility compute the values of the internal state (assuming that this is the right state machine), and apply the algorithm described above. We are guaranteed that the algorithm will succeed for the choice of the correct machine, and PAC arguments imply that a bad state machine is not likely to be chosen.

A high level description of the algorithm *Learn-PRS* is described in Figure 1. Using the above observations, we can apply Theorem 1 to show that PRS under these restrictions are learnable.

Theorem 2 *The algorithm Learn-PRS is a learn to act algorithm for the class of restricted PRS actions strategies.*

Hierarchical Strategies

The computing power of the PRS considered can be enhanced considerably if we allow a hierarchical construction. We consider such strategies where previously learned subroutines can be used as primitive actions in new strategies. In the full version of the paper we show that with annotation on the examples it is possible to learn hierarchical strategies, but that without annotation the task is hard even in propositional domains. For lack of space we just illustrate this point through an example.

A possible input for hierarchical learning problem is the subroutine $x_1 \rightarrow A; x_2 \rightarrow B;$ `True` $\rightarrow A$, with goal $x_4 = 1$, and where the priority is from left to right, and the two example runs: $R_1 = 01000, A, 01100, B, 10100, A, 11011$, and $R_2 = 10100, B, 11100, A, 11010, A, 10011$. The goal of the PRS being learned is to achieve $x_5 = 1$ which is indeed satisfied in the last state of the example runs.

Notice that if we know which actions were chosen by the subroutine we can rewrite the examples by slicing out the parts of the subroutine and replacing it with a new action, S. Using this new input, the greedy algorithm can find a consistent strategy. For example, using the information that for R_1 the second and third actions are taken by S, and for R_2 the second action is taken by S, it is easy to see that the PRS $\overline{x_3} \rightarrow A; x_2 \rightarrow S; x_1 \rightarrow B$ is consistent with the example runs. However, without this information the problem is NP-Hard; the hardness of the problem essentially lies in determining a proper annotation for the examples.

Conclusions

We presented a new framework for studying the problem of acting in dynamic stochastic domains. Following the learning to reason framework we stressed the importance of learning for accomplishing such tasks. The new approach combines features from several previous works, and in a sense narrows the gap between the reactive and declarative approaches. We have shown that results from the PAC model of learning can be generalized to the new framework, and thus many results hold in the new model as well. We also demonstrated that powerful representations in the form of production rule systems can be learned.

We have also shown that bias can play an important role in predicate invention; by using an appropriate bias, and grading the learning experiments in their difficulty, a new set of useful predicates (which are not available for introspection) can be learned and then used in the future.

Our model shares some ideas with reinforcement learning and EBL, and several related questions are raised by the new approach. In particular we believe that introducing bias in either of these frameworks can enhance our understanding of the problems. Other interesting questions concern learning of randomized strategies, and learning from "exercises" (Natarajan 1989), in the new model. Further discussion can be found in the full version of the paper.

Acknowledgments

I am grateful to Les Valiant for many discussions that helped in developing these ideas, and to Prasad Tadepalli for helpful comments on an earlier draft. This work was supported by ARO grant DAAL03-92-G-0115 and by NSF grant CCR-95-04436.

References

Anderson, J. 1983. *The Architecture of Cognition.* Harvard University Press.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's razor. *Information Processing Letters* 24:377–380.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69:165–204.

Cook, S. A. 1971. The complexity of theorem proving procedures. In *3rd annual ACM Symposium of the Theory of Computing*, 151–158.

DeJong, G., and Mooney, R. 1986. Explanation based learning: An alternative view. *Machine Learning* 1:145–176.

Fiechter, C. N. 1994. Efficient reinforcement learning. In *Proc. of Workshop on Comput. Learning Theory*, 88–97.

Gupta, N., and Nau, D. 1991. Complexity results for blocks world planning. In *Proceedings of AAAI-91*, 629–633.

Haussler, D. 1989. Learning conjunctive concepts in structural domains. *Machine Learning* 4(1):7–40.

Khardon, R., and Roth, D. 1994. Learning to reason. In *Proceedings of AAAI-94*, 682–687.

Khardon, R., and Roth, D. 1995. Learning to reason with a restricted view. In *Proc. Workshop on Comput. Learning Theory*, 301–310.

Khardon, R. 1995. Learning to take actions. Technical Report TR-28-95, Aiken Computation Lab., Harvard University.

Laird, J.; Rosenbloom, P.; and Newell, A. 1986. Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning* 1:11–46.

McCarthy, J. 1958. Programs with common sense. In Brachman, R., and Levesque, H., eds., *Readings in Knowledge Representation, 1985*. Morgan-Kaufmann.

Minton, S. 1990. Quantitative results concerning the utility of explanation based learning. *Artificial Intelligence* 42:363–391.

Mitchell, T.; Keller, R.; and Kedar-Cabelli, S. 1986. Explanation based learning: A unifying view. *Machine Learning* 1:47–80.

Natarajan, B. K. 1989. On learning from exercises. In *Proc. of Workshop on Comp. Learning Theory*, 72–87.

Rivest, R. L. 1987. Learning decision lists. *Machine Learning* 2(3):229–246.

Sutton, R. S. 1988. Learning to predict by the methods of temporal differences. *Machine Learning* 3(1):9–44.

Tadepalli, P., and Natarajan, B. 1996. A formal framework for speedup learning from problems and solutions. *Journal of AI Research.* Forthcoming.

Tadepalli, P. 1991. A formalization of explanation based macro-operator learning. In *Proceedings of IJCAI-91*, 616–622.

Tadepalli, P. 1992. A theory of unsupervised speedup learning. In *Proceedings of AAAI-92*, 229–234.

Valiant, L. G. 1985. Learning disjunctions of conjunctions. In *Proceedings of IJCAI-85*, 560–566.

Valiant, L. G. 1995. Rationality. In *Proc. Workshop on Comput. Learning Theory*, 3–14.

Weld, D. 1994. An introduction to least commitment planning. *AI magazine* 15(4):27–61.