A Theory of Unsupervised Speedup Learning

Prasad Tadepalli
Department of Computer Science
Oregon State University
Corvallis, OR 97331
(tadepalli@cs.orst.edu)

Abstract

Speedup learning seeks to improve the efficiency of search-based problem solvers. In this paper, we propose a new theoretical model of speedup learning which captures systems that improve problem solving performance by solving a user-given set of problems. We also use this model to motivate the notion of "batch problem solving," and argue that it is more congenial to learning than sequential problem solving. Our theoretical results are applicable to all serially decomposable domains. We empirically validate our results in the domain of Eight Puzzle.[1]

Introduction

Speedup learning seeks to improve the efficiency of search-based problem solvers. While the theory for concept learning is well-established, (for example, see [Valiant 1984; Natarajan 1991]), the theory of speedup learning is rapidly evolving [Cohen 1989; Elkan & Greiner 1991; Etzioni 1990; Greiner 1989; Laird 1990; Natarajan & Tadepalli 1988; Natarajan 1989; Subramanian & Feldman 1990; Tadepalli 1991a, etc.]. In this paper, we propose a formal model of speedup learning which views learning as jumps in the average asymptotic complexity of problem solving. The problem solving is "slow" in the beginning. But as the problem solver gains experience, its average asymptotic complexity reduces; thus it converges to a "fast" problem solver.

The model we propose is based on and extends our previous work. In our earlier work, reported in [Natarajan & Tadepalli 1988] and [Tadepalli 1991a], learning begins with a teacher-supplied set of problems and solutions. In these models, examples provide two kinds of information to the learner: First, they provide the problem distribution information, i.e., they tell the learner which problems are likely to occur in the world. Second, they also provide the solutions to these problems. Although this makes learning tractable, it is burdensome to the teacher in that she is required to solve the problems before giving them to the learner. In the new model, the teacher is only required to provide the problems. Since the learner has access to a complete and correct, if inefficient, problem solver, it can solve these problems by itself, while also learning a more efficient problem solver for the given distribution of problems.

The price for this decreased burden on the teacher is an increased complexity of learning. Hence it can be described as "unsupervised learning," and it captures the kind of learning attempted by problem solving and learning architectures like SOAR and PRODIGY [Laird, Rosenbloom & Newell 1986; Minton 1990]. However, most learning/problem solving architectures including the above assume sequential problem solving in that they do not attempt to solve a second problem until they have fully solved the first problem. In this paper we lift this sequential constraint on the problem solver and allow the system to solve a sample of problems in any way it is convenient. We call this "batch problem solving." We show that there are some important advantages to batch problem solving in that it can effectively take advantage of the multiple examples to discover regularities in the domain structure, which are otherwise difficult to uncover.

We show that earlier results by Korf in learning problem solving in domains like the Rubik's Cube can be explained in this framework [Korf 1985]. In particular, we generalize Korf's results and present algorithms that not only learn macro-tables but also learn the order in which the macro-operators might be used. We precisely define what it means to learn in this model and characterize the conditions under which this kind of learning is possible. We support our theoretical results with experiments.

Previous Work

This work is motivated by the experimental systems in Explanation-Based Learning (EBL), and the need to theoretically justify and explain their learning behavior [Laird, Rosenbloom & Newell 1986; Dejong & Mooney 1986; Minton 1990; Mitchell, Keller & Kedar-Cabelli

[1] This research was partially supported by the National Science Foundation under grant number IRI:9111231.

1986, etc.]. Our model of speedup learning draws its inspiration from "Probably Approximately Correct" (PAC) learning introduced by Valiant, which captures effective concept learning from examples [Valiant 1984; Natarajan 1991].

Our work is similar in spirit to many evolving models of speedup learning. For example, Cohen analyzes a "Solution Path Caching" mechanism and shows that whenever possible, it always improves the performance of the problem solver with a high probability [Cohen 1989]. Unlike in our work, however, the improved problem solver is not necessarily efficient. Greiner and Likuski formalize speedup learning as adding redundant learned rules to a horn-clause knowledge base to expedite query-processing [Greiner & Likuski 1989]. Subramanian and Feldman analyze an explanation-based learning technique in this framework, and show that it does not work very well when recursive proofs are involved [Subramanian & Feldman 1990]. Etzioni uses a complexity theoretic argument to arrive at a similar conclusion [Etzioni 1990]. Our results show that a polynomial-time problem solver can be learned under some conditions even when the "proof" or the problem solving trace has a recursive structure. This is consistent with the positive results achieved by speedup learning systems like SOAR in domains like Eight Puzzle [Laird, Rosenbloom & Newell 1986].

Our framework also resembles that of [Natarajan 1989] in that both of these models learn from problems only. Natarajan's model requires the teacher to create a nice distribution of problems, so called "exercises," which reflects the population of subproblems present in the given problems. Our model does not have this requirement, but relies on exponential-time search to build its own solutions.

PAC Speedup Learning Framework

The framework we introduce here is most similar to that of [Natarajan & Tadepalli 1988]. The main difference between the two is that in the current framework, unlike the previous one, the teacher only gives random problems to the learner without solutions. In this sense, this is "unsupervised learning."

The key feature of a speedup learning system is that it has access to a "domain model" or a "domain theory," which includes goal and operator models. This domain theory is also complete in that it can in principle be used to solve any problem using exhaustive search. The task of the learner is to take such a domain theory and a random sample of problems and output an efficient problem solver, which solves any randomly chosen problem with a high probability. As in the PAC model, we require that the practice problems are chosen from the same distribution as the test problems.

A *problem domain* D is a tuple $\langle S, G, O\rangle$, where

- S = A set of *states* or *problems*.
- G = A *goal* which is described as a set of conjunctive subgoals $\{g_1, g_2, \ldots\}$, where each subgoal g_i is a polynomial-time procedure that recognizes a set of states in S that satisfy that subgoal.
- O = A set of *operators* $\{o_1, o_2, \ldots\}$, where each o_i is a polynomial-time procedure that implements a partial function from S to S.

The combination of the goal G and the operators O is called a *theory* of D. Multiple goals can be accommodated into the above model by changing the state description to have two components: the current state and the goal. The operators manipulate only the current state, and G compares the current state to the goal description and determines if it is satisfied.

We assume that if an operator is applied to a state in which it is not applicable, it results in a distinguishable "dead state." We denote the result of applying an operator o to a state s by $o(s)$. The size of the state or problem s is the number of bits in its description, and is denoted by $|s|$. For simplicity, we assume that the operators are length-preserving, i.e., $|s| = |o(s)|$.

Unlike the standard EBL approaches, our goals and operators are not parameterized. Although this might appear to be a serious limitation, it is not the case. To keep the cost of instantiation tractable, the number of parameters of an operator (or macro-operator) must be kept small. If so, it can be replaced by a small number of completely instantiated operators, and hence it reduces to our model. Another important deviation from the EBL domain theories is that our operators are opaque. In addition to being a more realistic assumption, this also makes it possible to efficiently represent the domain theory [Tadepalli 1991b].

A problem s is *solvable* if there is a sequence of operators $\beta = (o_1, \ldots, o_d)$, and a sequence of states $(s_0, \ldots, s_d)$, such that (a) $s = s_0$,(b) for all i from 1 to d, $s_i = o_i(s_{i-1})$, and (c) s_d satisfies the goal G.

In the above case, β is a *solution sequence* of s, and d is the *length* of the solution sequence β. We call the maximum of the shortest solution lengths of all problems of a given size n, the *diameter* of problems of size n.

A *problem solver* f for D is a deterministic program that takes as input a problem, s, and computes its solution sequence, if such exists.

A *meta-domain* Z is any set of domains.

A *learning algorithm* for Z is an algorithm that takes as input the theory of any domain $D \in Z$ and some number of problems according to any problem distribution P and computes as output an approximate problem solver for D (with respect to P).

The learning protocol is as follows: First, the domain theory is given to the learner. The teacher then selects an arbitrary problem distribution. The learning algorithm has access to a routine called PROBLEM. At each call, PROBLEM randomly chooses a problem in the input domain, and gives it to the learner. The learning algorithm must output an approximate problem solver with a high probability after seeing a reasonable number of problems. The problem solver need only be approximately correct in the sense that it may fail to produce correct solutions with a small probability.

Definition 1 *An algorithm A is a learning algorithm for a meta-domain Z, if for any domain $D \in Z$, and any choice of a problem distribution P which is non-zero on only solvable problems of size n,*

1. *A takes as input the specification of a domain $D \in Z$, the problem size n, an error parameter ϵ, and a confidence parameter δ,*
2. *A may call PROBLEM, which returns problems x from domain D, where x is chosen with probability $P(x)$. The number of oracle calls of A, and the space requirement of A must be polynomial in n, the diameter d of problems of size n, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and the length of its input.*
3. *With probability at least $(1-\delta)$ A outputs a problem solver f which is approximately correct in the sense that $\Sigma_{x \in \Delta} P(x) \leq \epsilon$, where $\Delta = \{x | f \text{ fails on } x\}$.*
4. *There is a polynomial R such that, for any problem solver f output by A, the run time of f is bounded by $R(n, d, \frac{1}{\epsilon}, \frac{1}{\delta})$.*

The number of oracle calls of the learning algorithm is its sample complexity. Note that we place no restrictions on the time complexity of the learning algorithm, although we require that its space complexity and the time complexity of the output problem solver must be polynomials. The speedup occurs because of the time complexity gap between the problem solving before learning and the problem solving after learning. We call this framework PAC Speedup Learning.

Serial Decomposability and Macro-tables

This section introduces some of the basic terminology that we will be using later.

Here we make the assumption that states are representable as vectors of n discrete valued features, where the maximum number of values of any feature is bounded by a polynomial $h(n)$. We represent the value of the i^{th} feature of a state s by $s(i)$, and write $s = \langle s(1), \ldots, s(n) \rangle$.

In Rubik's Cube, the variables are cubie names, and their values are cubie positions. In Eight Puzzle, the variables are tiles, and their values are tile positions. Note that the above assumption makes it difficult to represent domains with relations, e.g., the blocks world.

Following Korf, we say that a domain is *serially decomposable* if there is a total ordering on the set of features such that the effect of any operator in the domain on a feature value is a function of the values of only that feature and all the features that precede it [Korf 1985].

```
Procedure Solve (Problem s)
For i := 1 thru n
    begin
        j := s(F_i);
        solution := solution.M(j, i)
        s := Apply(s, M(j, i));
    end;
Return (solution);
```

Table 1: Korf's Macro-table Algorithm

Rubik's Cube is serially decomposable for any ordering of features (also called "totally decomposable"). In Eight Puzzle, the effect of an operator on any tile depends only on the positions of that tile and the blank. Hence Eight Puzzle is serially decomposable for any ordering that orders the blank as the first feature.

We assume that there is a single, fixed goal state g described by $\langle g(1), \ldots, g(n) \rangle$.

Assume that a domain is serially decomposable with respect to a feature ordering $F_1, \ldots, F_n$. A macro-table is a set of macros $M(j, i)$ such that if $M(j, i)$ is used in a solvable state s where the features F_1 thru F_{i-1} have their goal values, and the feature F_i has a value j, then the resulting state is guaranteed to have goal values $g(F_1), \ldots, g(F_i)$ for features F_1 thru F_i.

Definition 2 *A meta-domain Z satisfies a* serial decomposability bias *if for any domain D in Z, there is a feature ordering $O = F_1, \ldots, F_n$ such that, (a) D is serially decomposable for O, and (b) every problem which is reachable from any solvable state is also solvable.*

If a domain is serially decomposable for the feature ordering $F_1, \ldots, F_n$, then any move sequence that takes a state where the features F_1 thru F_{i-1} have their goal values and the feature F_i has a value j to any other state where the features F_1 thru F_i have their goal values can be used as a macro $M(j, i)$. The reason for this is that the values of features F_1 thru F_i in the final state only depend on their values in the initial state, and not on the values of any other features.

Theorem 3 (Korf) *If a meta-domain satisfies the serial decomposability bias, then it has a macro-table.*

If a full macro-table with appropriately ordered features is given, then it can be used to construct solutions from any initial state without any backtracking search [Korf 1985], by repeatedly picking the appropriate macro and applying it to the state. The algorithm shown in Table 1 would do just that, assuming that F_i represents the feature that corresponds to the i^{th} column in the macro-table.

Batch Learning of Macro-tables

In this section we describe our batch learning algorithm and prove its correctness.

```
Procedure Batch-Learn (Problem set S, Current
        Column i, Current feature ordering F,
            Current Macro-table M)
Candidate features CF := {1,...,n} - {F_j | 1 ≤ j < i};
If CF = Φ Then return (M, F);
For each problem p ∈ S
  For each feature f ∈ CF
    If there is a macro m = C(p(f), f) stored for f
      Then If m works for p Then do nothing
                              Else CF := CF - {f}
      Else C(p(f), f) := ID-DFS(p, F ⋃{f});
If CF = Φ Then return (Fail);
For each f ∈ CF
  Begin
    F_i := f
    Add the new column C(*,f) to the macro-table M
    S' := {s' | s' = Apply(s, C(s(f), f)), where s ∈ S}
    If Batch-Learn(S', i + 1, F ⋃{f}, M) succeeds
      Then return (M, F)
      Else restore S, M, and F to old values;
  End
Return (Fail) ;
```

Table 2: An Algorithm to Learn Macro-tables

Korf's learning program builds a macro-table by exhaustively searching for a correct entry for each cell in the table [Korf 1985]. Unlike in Korf's system, we do not assume that the feature ordering in the macro-table is known to the system. The program must learn the feature ordering along with the macro-table. One way to learn this would be to exhaustively consider all possible feature orderings for each example. However, that would involve a search space of size $O(n!)$.

Our batch learning algorithm considers all $n!$ orderings in the worst-case, but uses multiple examples to tame this space. We consider different feature orderings while we build the macro-table column by column. That is, we move to the i^{th} column in the macro-table only when we have a consistent feature F_{i-1} for the $i-1^{th}$ column, and a set of macros for it which can solve *all* problems in the sample. The reason for this is that it allows us to prune out bad feature orderings early on by testing the macros learned from one problem on the other problems.

In the algorithm of Table 2, CF is the set of candidate features for the next column i of the macro-table, and C is the set of candidate columns for each feature in CF. At any column i, for each candidate feature f and feature value v, $C(v,f)$ contains a macro that would achieve the goal values for $\{F_1,\ldots,F_{i-1}\}\bigcup\{f\}$ when the goal values for features in $\{F_1,\ldots,F_{i-1}\}$ have already been achieved and the current value of feature f is v. If at any given time, there is already an applicable macro in C for the problem under consideration, it is tested on that problem. If it does not work,

6 1 7	6 1 3	1 2 3
8 4 3	8 4 7	8 0 4
5 0 2	2 5 0	7 6 5
Problem 1	Problem 2	Goal

Figure 1: Two problems and the goal

then that feature cannot give rise to a serially decomposable ordering, and is immediately pruned. For a given feature f, if there is no applicable macro for the problem, then the set of subgoals that correspond to features $\{F_1,\ldots,F_{i-1}\}\bigcup\{f\}$ is solved by an iterative-deepening depth first search, and the macro is stored in C. If there is an extension to the current feature ordering which retains the serial decomposability, then the program would end up with a set of consistent features and corresponding candidate macro-columns. The program explores each such candidate by backtracking search until it finds a serially decomposable ordering of features and a consistent macro-table. After learning a macro-column, the program updates the sample by solving the corresponding subgoal for each problem in the sample and proceeds to the next column.

Example: Eight Puzzle

Let r, l, u, and d represent the primitive operators of moving a tile right, left, up, and down respectively in Eight Puzzle. Macros are represented as strings made up of these letters. For notational ease, features (tiles) are labeled from 0 to 8, 0 standing for the blank and i for tile i. The positions are numbered by the tile numbers in the goal state, which is assumed to be fixed. Two problems and the goal state are shown in Figure 1. Assume that the learner is building the very first column in the macro-table. Since this is the first column, the program considers all features from 0 thru 8 as potential candidates and considers bringing each feature to its goal value. By doing iterative-deepening depth first search (ID-DFS) on the first problem, the program determines that the macro-operator "*d*" would achieve subgoal 0, i.e., bring the blank to its goal position, the macro-operator "*drdl*" would achieve subgoal 1 and so on. Hence it stores these different macros in the potential first columns of the macro-table: $C(6,0)$ = "*d*", $C(2,1)$ = "*drdl*", $C(5,2)$ = "*drdlulurdldru*," and so on. While doing the same on the second problem, the program checks to see if there is a macro in the table which is applicable for any subgoal. Since the position of the tile 1 is the same (2) in both the examples, the same macro $C(2,1)$ must be able to get tile 1 to its goal position for both the problems, provided that tile 1 is one of the correct candidates for the first column. However, applying the macro $C(2,1)$ = "*drdl*" on the second problem does not succeed in getting the tile 1 to its destination. Hence tile 1 is ruled out as the first column in the macro-table, and the program proceeds to the other tiles.

Note how multiple examples were used to avoid unnecessary backtracking. If the problem solver were to solve the second problem only after it completely solved the first problem, it would not have known that ordering tile 1 as the first feature is a bad idea until much later. In our algorithm, a bad feature ordering would be quickly detected by one example or another. Although backtracking is still theoretically possible, given enough number of examples, our program is found not to backtrack. This illustrates the power of batch problem solving. •

We are now ready to prove the main theoretical result of this paper.

Theorem 4 *If (a) Z is a meta-domain that satisfies the serial decomposability bias, and (b) the number of distinct values for each feature of each domain in Z is bounded by a polynomial function h of the problem size n, then Batch-Learn is a learning algorithm for Z.*

Proof (sketch): The proof closely follows the proof of sample complexity of finite hypothesis spaces in PAC learning [Blumer et al., 1987].

Assume that after learning from m examples, the learning algorithm terminated with a macro-table T. A macro-table is considered "bad" if the probability of its not being able to solve a random problem without search is greater than ϵ. We have to find an upper bound on m so that the probability of T being bad is less than δ.

The probability that a particular bad macro-table solves m random training problems (without search) is $(1-\epsilon)^m$. This is the probability of our algorithm learning a particular bad macro-table from a sample of size m. Hence, for the probability of learning any bad macro-table to be less than δ, we need to bound $|B|(1-\epsilon)^m$ by δ, where B is the set of bad tables.

Note that a table is bad only if (a) some of the macros in the macro-table are not filled up by any example, or (b) the feature ordering of the table is incorrect. Since the total number of macros in the full table is bounded by $nh(n)$, the number of unfilled subsets of the macros is bounded by $2^{nh(n)}$. Since the number of bad orderings is bounded by $n!$, the total number of bad tables $|B|$ is bounded by $n!2^{nh(n)}$. Hence we require $n!2^{nh(n)}(1-\epsilon)^m < \delta$. This holds if $m > \frac{1}{\epsilon}\{n \ln n + nh(n)ln2 + \ln\frac{1}{\delta}\}$.

Since h is polynomial in n, the sample complexity is polynomial in the required parameters. The macro-table can be used to solve any problem in time polynomial in problem size n and the maximum length of the macro-operators, which is bounded by the diameter d (because ID-DFS produces optimal solutions). The space requirements of Batch-Learn and the macro problem solver are also polynomial in d and n. Hence Batch-Learn is a learning algorithm for Z. •

The above theorem shows that the batch learning algorithm exploits serial decomposability, which allows it to compress the potentially exponential number of solutions into a polynomial size macro-table. Note that the solutions produced by the problem solver are not guaranteed to be optimal. This is not surprising, because finding optimal solutions for the $N \times N$ generalization of Eight Puzzle is intractable [Ratner & Warmuth 1986]. However, the solutions are guaranteed to be within n times the diameter of the state space, because each macro is an optimal solution to a particular subgoal.

Experimental Validation

The batch learning algorithm has been tested in the domain of Eight Puzzle. An encouraging result is that our program was able to learn with far fewer number of examples than predicted by the theory. With ϵ and δ set to 0.1, and $n = h(n) = 9$, the theoretical estimate of the number of examples is around 780. However, our program was able to learn the full macro-table and one of the correct feature orderings with only 23 examples with a uniform distribution over the problems. As expected, the problem solving after learning was extremely fast, and is invariant with respect to the number of macros learned.

The difference in the number of examples can be attributed to the worst-case nature of the theoretical estimates. For example, knowing that only about half of the macro-table needs to be filled to completely solve the domain reduces the theoretical bound by about half. Knowing that there are many possible correct feature orderings (all orderings that order the blank as the first feature) would further reduce this estimate. The distribution independence assumptions of the theory also contribute to an inflated estimate of the number of training examples.

Discussion

Our paper introduced the notion of batch problem solving, which seems more amenable to learning than incremental problem solving. The idea of batch problem solving is very general and can be used in conjunction with other learning algorithms as well. We showed that it helps avoid unnecessary backtracking by using information from multiple problems. Backtracking is avoided when there are many correct feature orderings, as in the Eight Puzzle domain, as well as when there are only a few. In the case when there are many such orderings, the program will have many opportunities to find a correct one, and is likely to avoid backtracking. When there are only a few orderings, then most of the incorrect orderings are likely to be ruled out by discovering conflicts, once again avoiding backtracking. Even when an incorrect feature ordering is learned due to a skewed problem distribution, this might still probably result in an approximately correct problem solver, because the same distribution that was used in learning is also to be used in testing the problem solver.

Our algorithm has interesting consequences to systems that combine "empirical" and "explanation-based" learning methods. For example, both A-EBL of [Cohen 1992] and IOE of [Dietterich & Flann 1989] empirically generalize complete explanations. We characterize speedup learning as finding a tractable problem solver for a given problem distribution and a given bias. The best algorithm for a given bias might finely combine the features of "empirical" and "explanation-based" approaches. For example, it may be appropriate to empirically combine parts of the explanations of many examples, which might in turn help find the remaining parts of the explanations. This is exactly what our algorithm does, while exploiting the serial decomposability property of the domain. Just as there is no single general-purpose concept learning algorithm, there is also no effective general-purpose speedup learning algorithm. Our methodology points to building special-purpose learning algorithms that are specifically designed to exploit various structural properties of sets of problem domains. We aim to build a tool box of speedup learning algorithms, each algorithm implementing a particular bias.

Conclusions

This paper integrates work from a number of areas including EBL, PAC learning, and macro-operator learning. We introduced a new model of speedup learning, which extended our earlier work by placing the responsibility to solve the training problems on the learner. In this sense, it is "unsupervised." We presented a new, implemented algorithm that learns feature orderings and macros for serially decomposable domains, and showed it to be correct and effective. We also introduced the idea of batch problem solving which appears more powerful than sequential problem solving in the context of speedup learning systems.

Acknowledgments

I am indebted to Balas Natarajan for introducing me to theoretical machine learning and lending me his hand whenever I needed it. I thank Jonathan Gratch, Sridhar Mahadevan, and the reviewers of this paper for many helpful comments.

References

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

Cohen, W. Solution path caching mechanisms which provably improve performance. Technical Report DCS-TR-254, Rutgers University, 1989.

Cohen, W. Abductive explanation-based learning: A solution to the multiple inconsistent explanation problem. *Machine Learning*, 8, 1992.

Dejong, G. and Mooney, R. Explanation-based learning: A differentiating view. *Machine Learning*, 2, 1986.

Elkan, C. and Greiner, R. Measuring and improving the effectiveness of representations. In *Proceedings of IJCAI-91*, Morgan Kaufmann, San Mateo, CA, 1991.

Etzioni, O. Why PRODIGY/EBL works. In *Proceedings of AAAI-90*, MIT Press, Cambridge, MA, 1990.

Flann, N. and Dietterich, T. G. A study of explanation-based methods for inductive learning. *Machine Learning*, 4, 1989.

Greiner, R and Likuski, J. Incorporating redundant learned rules: A preliminary formal analysis of EBL. In *Proceedings of IJCAI-89*, Morgan Kaufmann, San Mateo, CA, 1989.

Korf, R. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26, 1985.

Laird, J. E., Rosenbloom, P. S., and Newell, A. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1, 1986.

Laird, P. and Gamble, E. Extending EBG to term-rewriting systems. In *Proceedings of AAAI-90*, MIT Press, Cambridge, MA, 1990.

Minton, S. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42, 1990.

Mitchell, T., Keller, R., and Kedar-Cabelli, S. Explanation-based generalization: A unifying view. *Machine Learning*, 1, 1986.

Natarajan, B., and Tadepalli, P. Two new frameworks for learning. In *Proceedings of Machine Learning Conference*, Morgan Kaufmann, San Mateo, CA, 1988.

Natarajan, B. On learning from exercises. In *Proceedings of Computational Learning Theory Conference*, Morgan Kaufmann, San Mateo, CA, 1989.

Natarajan, B. *Machine Learning: A Theoretical Approach.* Morgan Kauffman, 1991.

Ratner, D. and Warmuth, M. Finding a shortest solution for the N X N extension of the 15-PUZZLE is intractable. In *Proceedings of AAAI-86*, Morgan Kaufmann, San Mateo, CA, 1986.

Subramanian, D. and Feldman, R. The utility of EBL in recursive domain theories. In *Proceedings of AAAI-90*, MIT Press, Cambridge, MA, 1990.

Tadepalli, P. A formalization of explanation-based macro-operator learning. In *Proceedings of IJCAI-91*, Morgan Kaufmann, San Mateo, CA, 1991.

Tadepalli, P. Learning with Inscrutable Theories. In *Proceedings of International Machine Learning Workshop*, Morgan kaufmann, San Mateo, CA, 1991.

Valiant, L. G. A theory of the learnable. *Communications of the ACM*, 11, 1984.