Discrete Sequence Prediction and its Applications

Philip Laird*
AI Research Branch
NASA Ames Research Center
Moffett Field, California 94035 (U.S.A.)
laird@pluto.arc.nasa.gov

Abstract

Learning from experience to predict sequences of discrete symbols is a fundamental problem in machine learning with many applications. We present a simple and practical algorithm (TDAG) for discrete sequence prediction, verify its performance on data compression tasks, and apply it to problem of dynamically optimizing Prolog programs for good average-case behavior.

Discrete Sequence Prediction: A Fundamental Problem

A few fundamental learning problems occur so often in practice that basic algorithms for solving them are becoming important elements of the machine-learning toolbox. Among these problems are pattern classification (learning by example to partition input vectors into two or more classes), clustering (grouping a set of input objects into an appropriate number of sets of objects), and delayed-reinforcement learning (learning to associate actions with states so as to maximize the expected long-term reward).

Discrete sequence prediction (DSP) is another basic learning problem whose importance, in my view, has been overlooked by researchers outside the data-compression community. In the most basic version the input is an infinite stream of discrete symbols about which we assume very little. The task is to find regularities in the input so that our ability to predict the next symbol progresses beyond random guessing. Humans exhibit remarkable skills in such problems, unconsciously learning, for example, that after Mary's telephone has rung three times, her machine will probably answer it on the fourth ring, or that the word "incontrovertible" will probably be followed by the word "evidence." The fact that predictions from different individuals are usually quite similar is further evidence that DSP is a fundamental skill in the human learning repertory.

Consider some applications where DSP plays an important role:

- *Information-theoretic applications* rely on a probability distribution to quantify the amount of "surprise" (information) in sequential processes. For example, an adaptive file compression procedure reads through a file generating codes to represent the text using as few bits as possible. Each character is passed to a learning element that forms a probability distribution for the next character(s). As this prediction improves, the file is compressed by assigning fewer bits to encode more probable strings. Closely related to file compression are game-playing situations where the ability to anticipate the opponent's moves can increase a player's expected score.
- *Dynamic program optimization* is the task of reformatting a program into an equivalent one tuned for the distribution of problems that it actually encounters. As the program solves a representative sample of problems, the learning element examines its decisions and search choices in sequence. From the resulting information about program execution sequences one constructs an optimized version of the program with better average-case performance.
- *Dynamic buffering algorithms* go beyond simple heuristics like least-recently-used for swapping items between a small cache and a mass-storage device. By learning patterns in the way items are requested, the algorithm can retain an item in the cache or initiate an anticipatory fetch for one that is likely to be requested soon.
- *Adaptive human-machine interfaces* reverse the common experience whereby a human quickly learns to predict how a program (or ATM, automobile, etc.) will respond. Years ago, operating systems acquired type-ahead buffering as an efficiency mechanism for humans; if the system can likewise learn to anticipate the human's responses, it can work-ahead, offer new options that combine several steps into one step, and so on.
- *Anomaly detection systems* are important for identifying illicit or unanticipated use of a system. Such

*Supported in part by the National Science Foundation (INT-9008726).

tasks are difficult because what is most interesting is precisely what is hardest to recognize and predict.

Some AI researchers have approached DSP as a knowledge-based task, taking advantage of available knowledge to predict future outcomes. While a few studies have attacked sequence extrapolation/prediction directly, e.g., (Dietterich and Michalski, 1986), more often the problem has been an embedded part of a larger research task, e.g., (Lindsay *et al.*, 1980). One can sometimes apply to the DSP problem algorithms not originally intended for this task. For example, feedforward nets (a concept-learning technique) can be trained to use the past few symbols to predict the next, e.g., (Sejnowski and Rosenberg, 1987).

Data compression is probably the simplest application of sequence prediction, since the text usually fits the model of an input stream of discrete symbols quite closely. *Adaptive* data compression (Lelewer and Hirschberg, 1987) learns in a single pass over the text: as the program sees more text and its ability to predict the remaining text improves, it achieves greater compression. The most widely used methods for linear data compression have been dictionary methods, wherein a dictionary of symbol sequences is constantly updated, and the index of the longest entry in the dictionary that matches the current text forms part of the code.

Less familiar are recent methods that use directed acyclic graphs to construct Markovian models of the source, e.g. (Bell *et al.*, 1990; Blumer, 1990; Williams, 1988). Such models have the clearest vision of the learning aspects of the problem and as such are most readily extended to problems other than data compression. The TDAG algorithm, presented below, is based on the Markov tree approach, of which many variants can be found in the literature. TDAG can, of course, be used for text compression, but our design is intended more for online tasks in which sequence prediction is only part of the problem. One such task, program optimization, is the original motivation for this research.

The TDAG Algorithm

TDAG (Transition Directed Acyclic Graph) is a sequence-learning tool. It assumes that the input consists of a sequence of discrete, uninterpreted symbols and that the input process can be adequately approximated in a reasonably short time using a small amount of storage. That neither the set of input symbols nor its cardinality need be known in advance is an important feature of the design. Another is that the time required to receive the next input symbol, learn from it, and return a prediction for the next symbol is tightly controlled, and in most applications, bounded.

We develop the TDAG algorithm by successive refinement, beginning with a very simple but impractical learning/prediction algorithm and subsequently repairing its faults. First, however, let us provide some intuition for the algorithm.

Assume that we have been inputting symbols for some time and that we want to predict the next one. Suppose the past four symbols were "`m th`" (the blank is significant). Our statistics show that this four-symbol sequence has not occurred very often, but that the *three*-symbol sequence "` th`" has been quite common and followed by `e` 60% of the time, `i` 15% of the time, `r` and `a` each 10%, and a few others with smaller likelihoods. This can form the basis for a probabilistic prediction of the next symbol and its likelihood. Alternately we could base such a prediction on just the previous *two* characters "`th`", on the preceding character "`h`", or on none of the preceding characters by just counting symbol frequencies. Or we could form a weighted combination of all these predictions. If both speed and accuracy matter, however, we will probably do best to base it on the strongest conditioning event "` th`", since by assumption it has occurred enough times for the prediction to be confident.

Maintaining a table of suffixes is wasteful of storage since one symbol becomes part of many suffixes. We shall instead use a successor tree, linking each symbol to those that have followed it in context.

The Basic Algorithm. TDAG learns by constructing a tree that initially consists of only the root node, Λ. Stored with each node is the following information:

- `symbol` is the input symbol associated with the node. For the root node, this symbol is undefined.
- `children` is a list of the nodes that are successors (children) of this node.
- `in-count` and `out-count` are counters, explained below.

If ν is a node, the notation `symbol`(ν) means the value of the `symbol` field stored in the node ν, and similarly for the other fields. There is one global variable, `state`, which is a FIFO queue of nodes; initially `state` contains only the node Λ. For each input symbol x the learning algorithm (Fig. 1) is called. We obtain a prediction by calling *project-from* and passing as an argument the last node ν on the `state` queue for which `out-count`(ν) is "sufficiently" high, in the following sense.

Note that the `in-count` field of a node ν counts the number of times that ν has been placed on the `state` queue. This occurs if `symbol`(ν) is input while its parent node is on the `state` queue, and we say that *ν has been visited* from its parent. The `out-count` field of a node ν counts the number of times that ν has been replaced by one of its children on the `state` queue. If μ is a child of ν, the ratio `in-count`(μ)/`out-count`(ν) is the proportion of μ visits among all visits from ν to its children. It is an empirical estimate of the probability of a transition from the node ν to μ. The confidence in this probability increases rapidly as `out-count`(ν) increases, so we can use a minimum value for the `out-count` value to select the node to project from.

```
input($x$): /* $x =$ the next input symbol */
1. Initialize new-state:= $\{\Lambda\}$.
2. For each node $\nu$ in state,
     • Let $\mu :=$ make-child($\nu, x$). /* (See below) */
     • Enqueue $\mu$ onto new-state.
3. state:=new-state.

make-child($\nu, x$): /* create or update the child of $\nu$
labeled $x$ */
1. In the list children($\nu$), find or create the node $\mu$ with a
   symbol of $x$. If creating it, initialize both its count fields
   to zero.
2. Increment in-count($\mu$) and out-count($\nu$) each by one.

project-from($\nu$): /* Return a prob. distrib.*/
1. Initialize projection:= {}.
2. For each child $\mu$ in children($\nu$), add to projection the
   pair
   [symbol($\mu$), (in-count($\mu$)/out-count($\nu$))].
3. Return projection.
```

Figure 1: Basic algorithm.

As a simple example (Fig. 2), suppose the string "a b" is input to an empty TDAG. The result is a TDAG with four nodes:

- Λ, the root, with two children. The out-count is two.
- a, the child of Λ labeled a, created upon arrival of the symbol a. This node has been visited only once, so its in-count is 1. Its only child has been visited once (with the arrival of the b), so its out-count is also 1.
- b, the child of Λ labeled b, created upon arrival of the symbol b. Its in-count is 1, but since no character has followed b, it has no children and its out-count is 0.
- **ab**, the child of the node **a**. It was created upon arrival of the symbol b, so its symbol is b. The in-count is 1, and the out-count is 0.

The state queue now contains three nodes: Λ, **b**, and **ab** (in that order). These nodes represent the three possible conditional events upon which we can base a prediction for the next symbol: Λ, the null conditional event; **b**; the previous symbol b; and **ab**, the two previous symbols. If we project from Λ, the resulting distribution is $[(\mathtt{a}, 1/2), (\mathtt{b}, 1/2)]$. We cannot yet project from either of the other two nodes, since both nodes are still leaves. Our confidence in the projection from Λ is low because it is based on only out-count(Λ) $= 2$ events; this field, however, increases linearly with the arrival of input symbols, and our confidence in the predictions based on it grows very rapidly.

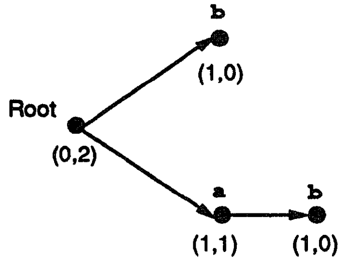

Figure 2: TDAG tree after "a b" input. The numbers in parentheses are, respectively, the in-count and out-count for the nodes.

The basic algorithm is impractical, in part because the number of nodes on the state queue grows linearly as the input algorithm continues to process symbols, and the number of nodes in the TDAG graph may grow quadratically. The trick, then, is to restrict the use of storage without corrupting or discarding the useful information.

The time for the procedure *input* to process each new symbol x (known as the *turnaround time*) is proportional to both the size of state and the time to search for the appropriate child of each state node (*make-child*, step 1). The improvements below will rectify the state-size problem; the search-time problem exists because there is no bound on the length of the children list. Therefore, to reduce the search for the appropriate child to virtually constant time, one should implement it using a hash table indexed by the node address ν and the symbol x, returning the address μ of the successor of ν labeled by x.

The Improved Algorithm. To make the basic algorithm practical, we shall make three modifications. Each change is governed by a parameter and requires that some additional information be stored at each node. It is also convenient to maintain a new global value **m** that increases by one for every input symbol. The changes are:

- *Bound the node probability.* We eliminate nodes in the graph that are rarely visited, since such nodes represent symbol strings that occur infrequently. For this purpose we establish a minimum threshold probability Θ and refuse to extend any node whose probability of occurring is below this threshold.
- *Bound the height of the TDAG graph.* The previous change in itself tends to limit the height of the TDAG graph, since nodes farther from the root occur less often on average. But there remain input streams that cause unbounded growth of the graph (for example, "a a a ..."). For safety, therefore, we introduce the parameter **H** and refuse to ex-

tend any node ν whose height height(ν) equals this threshold.

- *Bound the prediction size.* The time for *project-from* to compute a projection is proportional to K, the number of distinct symbols in the input. This is unacceptable for some real-time applications since K is unknown and in general may be quite large. Thus we limit the size of the projection to at most $\mathbf{P}$ symbols, $\mathbf{P} \geq 1$. Doing so means that any symbol whose empirical likelihood is at least $1/\mathbf{P}$ will be included in the projection.

The first change above is the most difficult to implement since it requires an estimate of $\Pr(\nu)$, the probability that ν will be visited on a randomly chosen round. Moreover, we can adopt either an *eager strategy* by extending a node until statistics indicate that $\Pr(\nu) < \Theta$ and then deleting its descendents, or a *lazy strategy* by refusing to extend a node until sufficient evidence exists that $\Pr(\nu) \geq \Theta$. Both strategies result ultimately in the same model. The eager strategy temporarily requires more storage but produces better predictions during the early stages of learning. In this paper we present the lazy strategy.

Note that in the basic algorithm the state always contains exactly one node of each height $h \leq \mathbf{m}$, where $\mathbf{m}$ is the number of input symbols so far. Let ν be a node of height h; with some reflection it is apparent that, if $\mathbf{m}$ is $\geq h$, then the fraction of times that ν has been the node of height h on state is $\Pr(\nu \mid \mathbf{m}) \equiv$ in-count(ν)$/(\mathbf{m} - h + 1)$. Moreover, as $\mathbf{m} \rightarrow \infty$, $\Pr(\nu \mid \mathbf{m})$ approaches $\Pr(\nu)$ if this limit exists. Since the decision about ν's extendibility must be made in finite time, however, we establish a parameter $\mathbf{N}$ and make the algorithm wait for $\mathbf{N}$ symbols (transitions from the root) before deciding the extendibility of nodes of height 1. Thereafter, another $2\mathbf{N}$ input symbols are required before nodes of height 2 are decided, and so on, with $h\mathbf{N}$ symbols needed to decide nodes of height h after deciding those of height $h - 1$. More symbols are needed for deciding nodes of greater height because the number of TDAG nodes with height h may be exponential in h; a sample size linear in h helps maintain a minimum confidence in our decision about each node, regardless of its height. Note that, with this policy, all nodes of height h become decidable after the arrival of $N(1 + 2 + \ldots + h) = Nh(h+1)/2$ symbols.

For the applications described in this paper a node, when marked extendible or unextendible, remains so thereafter, even if later the statistics seem to change. This policy is a deliberate compromise for efficiency. A switch extendible-p is stored with each node. It remains unvalued until a decision is reached as to whether ν is extendible, and then is set to true if and only if ν is extendible. (See the revised input algorithm in Figure 3.)

In the prediction algorithm, we store in each node a list most-likely-children of the $\mathbf{P}$ most likely children. Whenever an input symbol causes a node ν to be replaced by one of its children μ in the state, we adjust the list of ν's most likely children to account for the higher relative likelihood of μ. This can be done in time $\mathcal{O}(\mathbf{P})$. The algorithm is in Figure 4.

```
input(x): /* process one input symbol */
1. $\mathbf{m} := \mathbf{m} + 1$. Initialize new-state:= $\{\Lambda\}$.
2. For each node $\nu$ in state,
     • Let $\mu :=$ make-child($\nu, x$).
     • If extendible?($\mu$), then enqueue $\mu$ onto new-state.
3. state:=new-state.

make-child($\nu, x$): /* find or create a child node */
1. In the list children($\nu$) find or create the node $\nu$ with
   symbol($\mu$) = $x$. If creating it, initialize: in-count($\mu$)
   and out-count($\mu$) := 0, height($\mu$) := height($\nu$) + 1, and
   children($\mu$) and most-likely-children($\mu$) := empty.
2. Increment in-count($\mu$) and out-count($\nu$) each by one.
3. Revise the (ordered) list most-likely-children($\nu$) to
   reflect the increased likelihood of $\mu$.

Extendible?($\mu$): /*''Lazy'' Version */
1. If extendible-p($\mu$) is True or False, return its value.
2. Else let $h$ = height($\mu$); if $\mathbf{m} \leq \mathbf{N}h(h+1)/2$, then return
   False. ($\mu$ is still undecided).
3. If height($\mu$) = $\mathbf{H}$ (i.e., $\mu$ is at the maximum allowed
   height) or
   (in-count($\mu$) $- 1$) $< \Theta \cdot h\mathbf{N}$ (i.e., $\Pr(\mu)$ is below thresh-
   old), then
     • extendible-p($\mu$) := False.
     • Return False.
4. Else
     • extendible-p($\mu$) := True.
     • Return True.
```

Figure 3: Revised input algorithm.

Analysis

Space permits only the briefest sketch of the analysis of the correctness and complexity of the algorithm. The efficiency and space requirements are governed entirely by the four user parameters $\mathbf{H}$, Θ, $\mathbf{N}$, and $\mathbf{P}$. The *turnaround time* to process each input symbol is $\mathcal{O}(\mathbf{H} \log \mathbf{m})$. In many practical cases, where the input source does not suddenly and radically change its statistical characteristics, the $\mathcal{O}(\log \mathbf{m})$ factor can be eliminated by "freezing" the graph once the leaf nodes have all been marked unextendible; this occurs after at most $1 + (\mathbf{NH}(\mathbf{H}+1)/2)$ input symbols. The total number of TDAG nodes can be shown to be at most $K(1 + \mathbf{H}/\Theta)$, where K is the size of the input alphabet. Finally, the turnaround time of the prediction algorithm is $\mathcal{O}(\mathbf{P} \log \mathbf{m})$; again the $\mathcal{O}(\log \mathbf{m})$ factor is often removable in practice.

```
project-from($\nu$):
1. Initialize projection:= {}.
2. For each child $\mu$ in most-likely-children($\nu$), add to
   projection the pair
   [symbol($\mu$), in-count($\mu$)/out-count($\nu$)].
3. Return projection.
```

Figure 4: Revised prediction algorithm

Correctness and *usefulness* are distinct issues. Too many algorithms have been proven correct with respect to an arbitrary set of assumptions and yet turn out to be of little or no practical use. Conversely there are algorithms that appear to perform well without any formal correctness criteria, but the reliability and generality of such algorithms is problematic. Our TDAG design begins with specific performance requirements; hence usefulness has been the primary motivation. But a notion of "correctness" is also needed to ensure that the predictions have a well-defined meaning and to permit comparison with other algorithms.

Correctness is an extensional property that cannot be discussed without defining the family of input sources. Like many data compression algorithms the TDAG views the input as though it were generated by a stochastic deterministic finite automaton (SDFA) or Markov process. "Learning" an SDFA from examples is an intractable problem (Abe and Warmuth, 1990; Laird, 1988), and I am aware of no practical algorithm for learning general SDFA models in an online situation. The TDAG approach is to represent the SDFA as a Markov tree, in which the root node represents the SDFA in its steady state, depth-one nodes represent the possible one-step transitions from steady state, etc. It is not hard to prove that, *for any finite, discrete-time SDFA source S, the* input *algorithm of Figure 1 converges with probability one to the Markov probability tree for S. The predictions made by the* project-from *algorithm of Figure 1, with an input node ν of height h, converge to the h^{th}-step transition probabilities from steady state of S.*

The modifications to the basic TDAG version shown in Figures 3 and 4 determine how much of the Markov tree we retain and which nodes of the tree are suitable for prediction. Instead of shearing off all branches uniformly at a fixed height, the algorithm retains more nodes along branches that are most frequently traversed while cutting back the less probable paths. The parameters relate directly to the available computational resources (space and time), rather than to unobservable quantities like the number of states in the source process.

Of course, we can never be certain that the input process is really generated by an SDFA or that the parameter choices will guarantee convergence to a close approximation to the input process even when it is an SDFA. Usefulness is a property that can only be demonstrated, not proved.

Applications

Text compression is an easy, useful check of the quality of a DSP algorithm. The Huffman-code method of file compression (Lelewer and Hirschberg, 1987) uses the predicted probabilities for the next symbol to encode the symbols; the Huffman code assigns the fewest bits to the most probable characters, reserving longer codes for more improbable characters.

The TDAG serves nicely as the learning element in an adaptive compression program: each character is passed to the TDAG input routine and a prediction is returned for the next character. This prediction is used to build or modify a Huffman code, which is kept with each extendible node in the TDAG.

To decompress the file one uses the the inverse procedure: a Huffman code based on the prediction for the next character is used to decode the next character; that character then goes into the TDAG in return for a new Huffman code.

For compressing ASCII text, the TDAG parameters were set as follows: $\mathbf{H} = 15$ (though the actual graph never reached this height); $\mathbf{P} = 120$ (since no more than 120 characters actually occur in most ASCII text files); $\Theta = 0.002$; and $\mathbf{N} = 10$. The resulting program, while inefficient, gave compression ratios considerably better than those for the `compact` program (FGK algorithm) and, except for small files, better than those of the Unix `compress` utility (LZW algorithm). Sample results for files in three languages are shown in Figure 5.

	Size	Compression (%)		
Language	(bytes)	TDAG	`compress`	`compact`
C	4301	46.1	45.9	60.6
English	13334	51.7	53.2	60.3
Lisp	70101	33.4	38.0	41.5

Figure 5: Sample File Compression Results. Compression is the compressed length divided by the original length (smaller values are better).

Unfortunately, most DSP applications are not so straightforward. *Dynamic optimization* is the task of tuning a program for average-case efficiency by studying its behavior on a distribution of problems typical of its use in production. Sequences of computational steps that occur regularly can be partially evaluated and unfolded into the program, while constructs that entail search can be ordered to minimize the search time. Any program transformations, however, must result in a program that is semantically equivalent to the original. Explanation-based learning is a well-known example of a dynamic optimization method.

Adapting a DSP algorithm to perform dynamic optimization is non-trivial because prediction is only part of the problem. If several choices are possible, we must balance the likelihood of success against its cost. In repairing a car, for example, replacing a spark plug may be less likely to fix the problem than replacing the engine, but still worthwhile if the ratio of cost to probability of success is smaller.

I designed and wrote a new kind of dynamic optimizer for Prolog programs using a TDAG as the learning element. Details of the implementation are given elsewhere(Laird, 1992), along with a comparison to other methods. Here we summarize only the essential ideas. A Prolog compiler was modified in such a way that the compiled program passes its proof tree to a TDAG learning element along with measurements of the computational cost of refuting each subgoal. After running this program on a sample of several hundred typical problems, I used the resulting TDAG information to optimize the program. The predictions enable us to analyze whether any given clause-reordering or unfolding transformation will improve the average performance of the program. Both transformations leave the program semantics unchanged. Next, the newly optimized version of the program was recompiled with the modified compiler, and the TDAG learning process repeated, until no further optimizations could be found. The final program was then benchmarked against the original (unmodified) program.

As expected, the results depended on both the program and the distribution of problems. On the one hand a program for parsing a context-free language ran more than 40% faster as a result of dynamic optimization; this was mainly the result of unfolding recursive productions that occurred with certainty or near certainty in the sentences of the language. On the other hand a graph-coloring program coding a brute-force backtracking search algorithm was not expected to improve much, and, indeed, no improvement was obtained. Significantly, however, no performance degradation was observed either. Typical were speedups in the 10% to 20% range—which would be entirely satisfactory in a production application. See Figure 6 for sample results.

In general, the TDAG-based method enjoys a number of advantages over other approaches, e.g., the ability to apply multiple program transformations, absence of "generalization-to-N" anomalies, and a robustness due to the fact that the order of the examples has little influence on the final optimized program.

Conclusions

Discrete Sequence Prediction is a fundamental learning problem. The TDAG algorithm for the DSP problem is embarrassingly easy to implement and reason about, requires little knowledge about the input stream, has very fast turnaround time, uses little storage, is mathematically sound, and has worked well in practice.

Besides exploring new applications, I anticipate that future research directions will go beyond the current rote learning of high-likelihood sequences by generalizing from strings to patterns. This may help guide induction algorithms to new concepts and to ways to reformulate problems.

Program	Average Improvement (%)	
	CPU Time	Unifications
CF Parser	41.1	34.5
List Membership	18.5	17.2
Logic Circuit Layout	4.8	9.5
Graph 3-Coloring	0.20	−1.40

Figure 6: Sample Dynamic program optimization results.

Acknowledgments

Much of this work was done during my stay at the Machine Inference Section of the Electrotechnical Laboratory in Tsukuba, Japan. Thanks to the members of the laboratory, especially to Dr. Taisuke Sato. Thanks also to Wray Buntine, Peter Cheeseman, Oren Etzioni, Smadar Kedar, Steve Minton, Andy Philips, Ron Saul, Monte Zweben, and two reviewers for helpful suggestions. Peter Norvig generously supplied me with his elegant Prolog for use in the dynamic optimization research.

References

Abe, N. and Warmuth, M. 1990. On the computational complexity of approximating distributions by probabilistic automata. In *Proc. 3rd Workshop on Computational Learning Theory.*

Bell, T. C.; Cleary, J. G.; and Witten, I. H. 1990. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J.

Blumer, A. 1990. Application of DAWGs to data compression. In Capocelli, A., editor 1990, *Sequences: Combinatorics, Compression, Security, and Transmission.* Springer Verlag, New York. 303 – 311.

Dietterich, T. and Michalski, R. 1986. Learning to predict sequences. In al., R. S. Michalskiet, editor 1986, *Machine Learning: An AI Approach, Vol. II.* Morgan Kaufmann.

Laird, P. 1988. Efficient unsupervised learning. In Haussler, D. and Pitt, L., editors 1988, *Proceedings, 1st Comput. Learning Theory Workshop.* Morgan Kaufmann.

Laird, P. 1992. Dynamic optimization. In *Proc., 9th International Machine Learning Conference.* Morgan Kaufmann.

Lelewer, D. and Hirschberg, D. S. 1987. Data compression. *ACM Computing Surveys* 19:262 – 296.

Lindsay, R.; Buchanan, B.; and *et al.*, 1980. *DENDRAL.* McGraw-Hill, New York.

Norvig, P. 1991. *Paradigms of A.I. Programming: Case Studies in Common LISP.* Morgan Kaufmann.

Sejnowski, T. and Rosenberg, C. 1987. Parallel networks that learn to pronounce English text. *Complex Systems* 1:145–168.

Williams, R. 1988. Dynamic history predictive compression. *Information Systems* 13(1):129–140.