Small Journal Name, ??, 1–20 (??)

Associative Reinforcement Learning: Functions in k-DNF

LESLIE PACK KAELBLING LPK@CS.BROWN.EDU
Computer Science Department
Box 1910
Brown University
Providence, RI 02912-1910 USA

Editor:

Abstract. An agent that must learn to act in the world by trial and error faces the *reinforcement learning* problem, which is quite different from standard concept learning. Although good algorithms exist for this problem in the general case, they are often quite inefficient and do not exhibit generalization. One strategy is to find restricted classes of action policies that can be learned more efficiently. This paper pursues that strategy by developing algorithms that can efficiently learn action maps that are expressible in k-DNF. The algorithms are compared with existing methods in empirical trials and are shown to have very good performance.

Keywords: reinforcement learning, generalization, k-DNF

1. Reinforcement Learning

Consider an agent that must learn to act in the world. At each moment in time, it gets information about the world from its sensors and must choose an action to take. Having executed an action, the agent gets a signal from the world that indicates how well the agent is performing; we shall call this a *reinforcement* signal. The reinforcement signal can be binary or real-valued and it will typically be noisy.

This learning scenario is quite different from standard concept learning, in which a teacher presents the learner with a set of input/output pairs. In the reinforcement-learning scenario, the agent must choose an output to generate in response to each input. The reinforcement signal it receives indicates only how successful that output was; it carries no information about how successful other outputs might have been. In addition, the fact that the reinforcement signal is noisy means that each output will have to be generated a number of times in order for the agent to acquire an accurate picture of which is better. In reinforcement-learning situations, an agent may choose an action because it expects it to have good results; however, it may also choose an action in order to gain information about its expected results. The tradeoff between acting to gain reinforcement and acting to gain information, sometimes referred to as the problem of *exploration versus exploitation*, makes this problem especially interesting. The formal foundations of reinforcement learning have been widely studied (Berry & Fristedt, 1985; Kaelbling, 1993b; Narendra & Thathachar, 1989; Williams, 1992).

For simplicity of presentation, this paper will initially focus on a simple case of the reinforcement learning problem in which the following assumptions hold:

- the agent has only two possible actions
- the reinforcement signal at time $t + 1$ reflects only the success of the action taken at time t
- reinforcement received for performing a particular action in a particular situation is 1 with some probability p and 0 with probability $1 - p$, and each trial is independent
- the expected reinforcement value of doing a particular action in a particular input situation stays constant for the entire run of the learning algorithm

Section 7 discusses the extension of the results in this paper to situations in which each of the above assumptions is relaxed.

An agent's strategy can be seen as a mapping from its input space to its action space. In the case we are considering initially, the output space has only two elements, so the agent's action map can be thought of as a Boolean function.

Because we are interested in practical learning methods for agents in dynamic worlds, we will require our algorithms to be *strictly incremental*. A learning algorithm is strictly incremental if the time and space it takes to process each new input has a constant upper bound that is independent of the number of inputs seen so far. This keeps the program from slowing down over time which would, effectively, change the dynamics of the world for the agent.

2. Complexity, Efficiency, and Generalization

There are a number of good algorithms for the reinforcement-learning scenario we are interested in, including learning-automata algorithms (Narendra & Thathachar, 1989), Sutton's reinforcement-comparison methods (1984), and Kaelbling's interval-estimation methods (1993b). These algorithms were originally developed for the case in which the agent has no inputs other than reinforcement and merely needs to decide which action it should take all the time. They can be extended to the case of having many input situations simply by making a copy of the algorithm for each possible input situation. Such "copy" algorithms require space proportional to the number of inputs in the space; as we begin to apply copy algorithms to real-world problems, their time and space requirements will make them impractical. In addition, copy algorithms completely compartmentalize the information they have about individual input situations. If a copy algorithm learns to perform a particular action in one input situation, that knowledge has no influence on what it will do in similar input situations. In realistic environments, an agent cannot expect ever to encounter all of the input situations, let alone have enough experience with each one to learn the appropriate response. Thus, we must develop algorithms that will generalize across input situations.

It is important to note, however, that in order to find more efficient algorithms, we must give up something. What we will be giving up is the possibility of learning any arbitrary action mapping. In the worst case, the only way to represent a mapping is as a complete look-up table, which is what copy algorithms do. There are many useful and interesting functions that can be represented much more efficiently, and the remainder of this work will rest on the hope and expectation that an agent can learn to act effectively in interesting environments without needing action maps of pathological complexity.

Action maps with a single Boolean output can be described by formulae in propositional logic, in which the atoms are input bits. The formula $(i_1 \wedge i_2) \vee \neg i_0$ describes an action map that performs action 1 whenever input bits 1 and 2 are on or input bit 0 is off and performs action 0 otherwise. When there are only two possible actions, we can describe the class of action maps that are learnable by an algorithm in terms of syntactic restrictions on the corresponding class of propositional formulae. This method is widely used in the literature on computational learning theory.

A restriction that has proved useful to the concept-learning community is to the class of functions that can be expressed as propositional formulae in k-DNF. A formula is said to be in *disjunctive normal form* (DNF) if it is syntactically organized into a disjunction of purely conjunctive terms; there is a simple algorithmic method for converting any formula into DNF (Enderton, 1972). A formula is in the class k-DNF if and only if its representation in DNF contains only conjunctive terms of length k or less. There is no restriction on the number of conjunctive terms—just their length. Whenever k is less than the number of atoms in the domain, the class k-DNF is a restriction on the class of functions. Another restriction on Boolean functions is simply to limit the syntactic complexity (in terms of depth or number of symbols) of the simplest propositional formula expressing the function. This restriction is explored in a companion paper (Kaelbling, 1993a).

The artificial neural network community has also considered methods for learning restricted classes of functions. The simplest restriction is that of *linear separability*. A Boolean function is linearly separable if it is possible to divide the input space by an n-dimensional hyperplane such that all inputs for which 0 is the correct output are on one side and all inputs for which 1 is the correct output are on the other. A larger class of functions can be learned by layers of sigmoidal units; this class, parameterized by the number of units and the nature of their interconnections does not have another obviously intuitive description.

This paper explores techniques for constructing efficient algorithms for reinforcement learning that generalize over the input space. Section 3 presents the interval-estimation algorithm, which provides a good non-generalizing solution to the reinforcement-learning problem, and serves as a basis for some of of the new algorithms in this paper. In section 4 we outline existing artificial neural-network methods. Section 5 presents two new methods based on the restriction to k-DNF. In each case, we consider the computational complexity and representational capacity of the methods. In section 6 we present the results of empirical testing of all the methods described herein in a set of environments that individually stress different aspects of learning. Finally, we consider the consequences of relaxing the assumptions made initially and investigate directions for future work.

The algorithms presented in this paper are described in a standard form consisting of three components: s_0 is the initial internal state of the algorithm; $u(s, i, a, r)$ is the update function, which takes the state of the algorithm s, the last input i, the last action a, and the reinforcement value received r, and generates a new algorithm state; and $e(s, i)$ is the evaluation function, which takes an algorithm state s and an input i, and generates an action.

3. Interval Estimation Method

The interval estimation method is a simple statistical algorithm for reinforcement learning. By allowing the state of the algorithm to encode not only estimates of the relative merits of the various actions, but also the degree of confidence that we have in those estimates, the interval estimation method makes it easier to control the tradeoff between acting to gain information and acting to gain reinforcement.

The basic interval estimation algorithm is specified in figure 1. The state consists of simple statistics: for each action a, n_a and x_a are the number of times that the action has been executed and the number of those times that have resulted in reinforcement value 1, respectively. The evaluation function uses these statistics to compute, for each action, a confidence interval on the underlying probability, p_a, of receiving reinforcement value 1 given that action a is executed. If n is the number of trials and x the number of successes arising from a series of Bernoulli trials with probability p, the upper bound of a $100(1-\alpha)\%$ confidence interval for p can be approximated by $ub(x, n)$. [1] The evaluation function chooses the action with the highest upper bound on expected reinforcement.

Initially, each of the actions will have an upper bound of 1, and action 0 will be chosen arbitrarily. As more trials take place, the bounds will tighten. The interval estimation method balances acting to gain information with acting to gain reinforcement by taking advantage of the fact that there are two reasons that the upper bound for an action might be high: because there is little information about that action, causing the confidence interval to be large, or because there is information that the action is good, causing the whole confidence interval to be high. The parameter $z_{\alpha/2}$ is the value that will be exceeded by the value of a standard normal variable with probability $\alpha/2$. It controls the size of the confidence intervals and, thus, the relative weights given to acting to gain information and acting to gain reinforcement. As α increases, more instances of reinforcement value 0 are required to drive down the upper bound of the confidence intervals, causing more weight to be placed on acting to gain information. By the DeMoivre-Laplace theorem (Larsen & Marx, 1986), these bounds will converge, in the limit, to the true underlying probability values, and, hence, if each action is continually attempted, this algorithm will converge to the optimal strategy. This algorithm is discussed in much greater detail elsewhere (Kaelbling, 1993b).

To apply the IE algorithm to domains with multiple inputs, we can simply make a copy of the basic algorithm for each individual input.

Algorithm IE
The initial state, s_0, consists of the integer variables x_0, n_0, x_1, and n_1, initialized to 0.

$u(s, a, r) =$ if $a = 0$ then begin
    $x_0 := x_0 + r$
    $n_0 := n_0 + 1$
  end else begin
    $x_1 := x_1 + r$
    $n_1 := n_1 + 1$
  end
$e(s) =$ if $ub(x_0, n_0) > ub(x_1, n_1)$ then
    return 0
  else
    return 1

where

$$ub(x, n) = \frac{\frac{x}{n} + \frac{z^2_{\alpha/2}}{2n} + \frac{z_{\alpha/2}}{\sqrt{n}}\sqrt{\left(\frac{x}{n}\right)\left(1 - \frac{x}{n}\right) + \frac{z^2_{\alpha/2}}{4n}}}{1 + \frac{z^2_{\alpha/2}}{n}}$$

and $z_{\alpha/2} > 0$.

Figure 1. The interval estimation (IE) algorithm.

4. Artificial Neural Network Techniques

4.1. Linear Associators

Most early reinforcement-learning algorithms were based on a binary model of reinforcement, in which actions were either good or bad. Good actions were strengthened and bad ones inhibited. Even with binary reinforcement, these algorithms tended to perform poorly when, for example, one action succeeded with probability 0.2 and the other with probability 0.1, since both actions were inhibited most of the time. This problem was addressed by Barto, Sutton, and Brouwer (1981) in the ASN, by estimating the average reinforcement for each input and strengthening actions according to the different between the reinforcement value actually received and the average reinforcement. In the previous scenario, this would have the effect of generating large changes for positive reinforcements (actual value 1 minus expected value 0.2) and smaller changes for negative reinforcements (actual value 0 minus expected value 0.2).

This is the notion that underlies the reinforcement-comparison algorithms, which include a component that learns a mapping from inputs to estimated reinforcement values, p. Actions are then rewarded or punished according to the difference between the actual reinforcement received and the estimated reinforcement value of the state, $r - p$.

Algorithm LARC
The input is represented as an $M+1$-dimensional vector i, in which the last component is set to a constant value. The internal state, s_0, consists of two $M+1$-dimensional vectors, $\mathbf{v}$ and $\mathbf{w}$, initialized to small random values.

$$
\begin{aligned}
u(s,i,a,r) = \quad & \text{let } p := \mathbf{v}\cdot i \\
& \text{for } j = 1 \text{ to } M+1 \text{ do begin} \\
& \qquad w_j := w_j + \alpha(r-p)(a-1/2)i_j \\
& \qquad v_j := v_j + \beta(r-p)i_j \\
& \text{end}
\end{aligned}
$$

$$
e(s,i) = \begin{cases} 1 & \text{if } \mathbf{w}\cdot i + \nu > 0 \\ 0 & \text{otherwise} \end{cases}
$$

where $\alpha > 0$, $0 < \beta < 1$, and ν is a normally distributed random variable of mean 0 and standard deviation δ_y.

Figure 2. The linear-associator reinforcement-comparison (LARC) algorithm.

Reinforcement-comparison methods are crucial for use in domains with multiple-valued or real-valued reinforcement, because there is no way to estimate the utility of an action without having an estimation of how good actions are, in general. It is useful to note that the interval estimation algorithm, and other algorithms based on similar statistical techniques, also build a reinforcement model, which is implicit in the internal data structures of the algorithm.

Sutton (1984) gives methods for converting standard reinforcement-learning algorithms to work in an associative setting, allowing an agent to learn efficiently and to generalize across input states. He uses a version of the Widrow-Hoff or Adaline (Widrow & Hoff, 1960) weight-update algorithm to associate different internal state values with different input situations. This approach is illustrated by the LARC (linear-associator reinforcement-comparison) algorithm shown in figure 2.

The inputs to the algorithm are represented as $M+1$-dimensional vectors, with the input functioning as an adjustable threshold. The output, $e(s,i)$, has value 1 or 0 depending on the dot product of the weight vector $\mathbf{w}$ and i and on the value of the random variable ν. The updating of the vector $\mathbf{w}$ is somewhat complicated: each component is incremented by a value with four terms. The first term, α, is a constant that represents the learning rate. The next term, $r-p$, represents the difference between the actual reinforcement received and the predicted reinforcement, p. The predicted reinforcement, p, is generated using a standard linear associator that learns to associate input vectors with reinforcement values by setting the weights in vector $\mathbf{v}$. The third term in the update function for $\mathbf{w}$ is $a-1/2$: it has constant absolute value and the sign is used to encode which action was taken; when $r > p$, the action taken is made more likely and when $r < p$, the action not taken is made more likely. The final term is i_j, which causes the jth component of the weight vector to be adjusted in proportion to the jth value of the input.

The space required for the state, as well as time for both update and evaluation operations is $O(M)$, where M is the number of input bits. If the input set is encoded by bit strings, the linear-associator approach can achieve an exponential improvement in space over the copy approach, because the size of the state of the linear-associator is proportional to the number of input bits rather than to the number of inputs. This algorithm works well on simple problems, but algorithms of this type are incapable of learning functions that are not linearly separable (Minsky & Papert, 1969), and do not necessarily succeed on all separable problems.

4.2. Error Backpropagation

To remedy the limitations of the linear-associator approach, multi-layer neural-network learning methods have been adapted to reinforcement learning. Anderson (1986), Werbos (1988), and Munro (1987), among others, have used error back-propagation methods (Hertz et al., 1991) with hidden units in order to allow reinforcement-learning systems to learn more complex action mappings. Williams (1988) presents an analysis of the use of back-propagation in associative reinforcement-learning systems. He shows that a class of reinforcement-learning algorithms that use back-propagation (an instance of which is given below) perform gradient ascent search in the direction of maximal expected reinforcement. This technique is effective and allows considerably more generalization across input states, but it requires many more presentations of the data in order for the internal units to converge to the features that they need to detect in order to compute the overall function correctly.

As an example of the application of error back-propagation methods to reinforcement learning, Anderson's method (1986) will be examined in more detail. It uses two networks: one for learning to predict reinforcement and one for learning which action to take. The weights in the action network are updated in proportion to the difference between actual and predicted reinforcement, making this an instance of the reinforcement-comparison method (discussed in section 4.1 above). Each of the networks has two layers, with all of the hidden units connected to all of the inputs and all of the inputs and hidden units connected to the outputs. The system was designed to work in worlds with delayed reinforcement, but it is easily simplified to work in domains with instantaneous reward.

The BPRC algorithm, which is analogous to the LARC algorithm, using two-layer fully-connected backpropagation networks to store the mapping from inputs to expected reinforcement values and from inputs to actions, is described in figure 3 and is explained in greater detail by Anderson. The presentation here is simplified in a number of respects, however. In this version, there is no use of momentum and the term $(a - 1/2)$ is used to indicate the choice of action rather than the more complex expression used by Anderson. Also, Anderson uses a different distribution for the random variable ν.

This method is theoretically able to learn very complex functions, but tends to require many training instances before it converges. The time and space complexity for this algorithm is $O(MH)$, where M is the number of input bits and H is the number of hidden units. Also, this method is somewhat less robust than the more standard version of error back-propagation that learns from input/output pairs, because the error signal generated by the reinforcement-learning system is not always correct. In addition, the two networks

Algorithm BPRC
The input is represented as an $M + 1$-dimensional vector i, in which the last element contains a constant value. The internal state, s_0, consists of the internal states for two two-layer, fully-connected backpropagation networks, each with H hidden units and a single output unit.

$u(s, i, a, r) =$ Present i to the evaluation network, generating p as output
Train the evaluation network with input i and error $r - p$
Train the action network with input i and error $(r - p)(\alpha - 1/2)$

$e(s, i) =$ Present i to the action network, generating v_a as output
return $\begin{cases} 1 & \text{if } v_a + \nu > 0 \\ 0 & \text{otherwise} \end{cases}$

The learning rates are β and β_h for the output and hidden units of the evaluation network and ρ and ρ_h for the output and hidden units of the action unit. The squashing function for the hidden units is $f(x) = 1/(1 + e^{-x})$, and ν is a normally distributed random variable of mean 0 and standard deviation δ_y.

Figure 3. An application of error-backpropagation to reinforcement learning.

must converge simultaneously to the appropriate solutions; if the learning rates are not set correctly, the system can converge to a state in which the evaluation network decides that all input states have a very poor expected performance, which is in fact true, given the current state of the action network. In such a situation, the weights will not be updated and the system becomes stuck.

5. Learning Mappings in k-DNF

Valiant was one of the first to consider the restriction to learning functions expressible in k-DNF (Valiant, 1984; Valiant, 1985), and considerable work in the computational learning theory community has followed from this. There have been objections to the choice of k-DNF as a bias for a learning algorithm, mostly based on "artificiality" or "syntacticness." It is true that the bias is described syntactically, but the choice of the set of functions in 4-DNF, for example, is no less artificial than the choice of the set of linearly separable functions or the set of functions that can be represented with one sigmoidal hidden unit. There is no basis for choosing a bias *a priori* unless something is known about the the functions to be learned (Wolpert, 1993). Thus, it is important to investigate and develop algorithms for a variety of different biases, then to be careful, when applying them, to choose appropriately.

One motivation for studying the bias of k-DNF, in particular, is that empirical studies of concept learning in humans have shown that the number of conjunctive items that must be taken together (that is, the k in k-DNF) is highly indicative of the difficulty of the concept learning task (Gluck, 1991). These results do not argue that k-DNF is the best bias for technical reasons, but do argue for its plausibility.

Valiant developed an algorithm, shown below, for learning functions in k-DNF from input-output pairs, which only uses the input-output pairs with output 0.

Algorithm VALIANT

Let T be initialized to the set of conjunctive terms of length k over the set of atoms (corresponding to the input bits) and their negations, and let L be the number of learning instances required to learn the concept to the desired accuracy.[2]

```
for i := 1 to L do begin
        v := randomly drawn negative instance
        T := T− any term that is satisfied by v
end
return T
```

The VALIANT algorithm returns the set of terms remaining in T, with the interpretation that their disjunction is the concept that was learned by the algorithm. This method simply examines a fixed number of negative instances and removes any term from T that would have caused one of the negative instances to be satisfied.[3]

5.1. Combining the LARC *and* VALIANT *Algorithms*

Given our interest in restricted classes of functions, we can construct a hybrid algorithm for learning action maps in k-DNF. It hinges on the simple observation that any such function is a linear combination of terms in the set T, where T is the set of conjunctive terms of length k over the set of atoms (corresponding to the input bits) and their negations. It is possible to take the original M-bit input signal and transduce it to a wider signal that is the result of evaluating each member of T on the original inputs. We can use this new signal as input to a linear-associative reinforcement learning algorithm, such as Sutton's LARC algorithm (described in figure 2). If there are M input bits, the set T has size $\binom{2M}{k}$ because we are choosing from the set of input bits and their negations. However, we can eliminate all elements that contain both an atom and its negation, yielding a set of size $2^k\binom{M}{k}$. The combined algorithm, called LARCKDNF, is described formally in figure 4 and schematically in figure 5. This method is reminiscent of Rosenblatt's original approach (Rosenblatt, 1961) of building networks with a large set of random fixed features in the first layer.

The space required by the LARCKDNF algorithm, as well as the time to update the internal state or to evaluate an input instance, is proportional to the size of T, and thus, $O(M^k)$.

5.2. Interval Estimation Algorithm for k-DNF

The interval estimation algorithm for k-DNF is, like the LARCKDNF algorithm, based on Valiant's algorithm, but the interval estimation algorithm uses standard statistical estimation methods, like those used in the IE algorithm, rather than weight adjustments.

Algorithm LARCKDNF

Let F_T be a function mapping an M-bit input vector into a $2^k\binom{M}{k}$-bit vector, each of whose elements is the result of evaluating an element of T on the raw input vector.
Let s_0 of this algorithm be the initial state, s_0, of an instance of the LARC algorithm with $2^k\binom{M}{k}$ bits. The update function will be u of LARC, with the input $F_T(i)$, and, similarly, the evaluation will be e of LARC, with the input $F_T(i)$.

Figure 4. The linear-association reinforcement-comparison algorithm for learning functions in k-DNF from reinforcement.

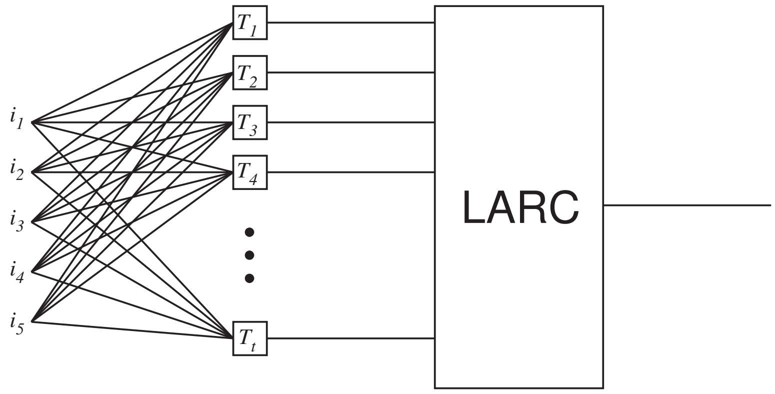

Figure 5. The LARCKDNF algorithm constructs all of the k-wide conjunctions over the inputs and their negations, then feeds them to an instance of the LARC algorithm.

The algorithm will first be described independent of particular statistical tests, which will be introduced later in this section. We shall need the following definitions, however. An input bit vector *satisfies* a term whenever all the bits mentioned positively in the term have value 1 in the input and all the bits mentioned negatively in the term have value 0 in the input. The quantity $er(t, a)$ is the expected value of the reinforcement that the agent will gain, per trial, if it generates action a whenever term t is satisfied by the input and generates action $\neg a$ otherwise. The quantity $ubr_\alpha(t, a)$ is the upper bound of a $100(1-\alpha)\%$ confidence interval on the expected reinforcement gained from performing action a whenever term t is satisfied by the input and action $\neg a$ otherwise. The formal definition of the algorithm is given in figure 6.

As in the regular interval estimation algorithm, the evaluation criterion is chosen in such a way as to make the important trade-off between acting to gain information and acting to gain reinforcement. Thus, the first requirement for a term to cause a 1 to be generated is that the upper bound on the expected reinforcement of generating a 1 when this term is satisfied is higher than the upper bound on the expected reinforcement of generating a 0 when the term is satisfied.

Algorithm IEKDNF

$s_0 =$ the set T, with a collection of statistics
associated with each member of the set

$e(s, i) =$ for each t in s
    if i satisfies t and
        $ubr_\alpha(t, 1) > ubr_\alpha(t, 0)$ and
        Pr$(er(t, 1) = er(t, 0)) < \beta$
    then return 1
  return 0

$u(s, i, a, r) =$ for each t in s
    update_term_statistics(t, i, a, r)
  return s

Figure 6. The IEKDNF algorithm for learning concepts in k-DNF from reinforcement.

Let the *equivalence probability* of a term be the probability that the expected reinforcement is the same no matter what choice of action is made when the term is satisfied. The second requirement for a term to cause a 1 to be generated is that the equivalence probability be small. Without this criterion, terms for which no action is better will, roughly, alternate between choosing action 1 and action 0. Because the output of the entire algorithm will be 1 whenever any term has value 1, this alternation of values can cause a large number of wrong answers. Thus, if we can convince ourselves that a term is irrelevant by showing that its choice of action makes no difference, we can safely ignore it.

At any moment in the operation of this algorithm, we can extract a symbolic description of its current action function. It is the disjunction of all terms t such that $ubr_\alpha(t, 1) > ubr_\alpha(t, 0)$ and Pr$(er(t, 1) = er(t, 0)) < \beta$. This is the k-DNF expression according to which the agent is choosing its actions.

In the simple Boolean reinforcement-learning scenario, the necessary statistical tests are quite simple. For each term, the following statistics are stored: n_0, the number of trials of action 0; s_0, the number of successes of action 0; n_1, the number of trials of action 1; and s_1, the number of successes of action 1. These are incremented only when the associated term is satisfied by the current input instance. Using the definition of $ub(x, n)$ from figure 1, we can define $ubr_\alpha(t, 0)$ as $ub(s_0, n_0)$ and $ubr_\alpha(t, 1)$ as $ub(s_1, n_1)$, where s_0, n_0, s_1, and n_1 are the statistics associated with term t and α is used in the computation of ub.

To test for equality of the underlying Bernoulli parameters, we use a two-sided test at the β level of significance that rejects the hypothesis that the parameters are equal whenever

$$\frac{\frac{s_0}{n_0} - \frac{s_1}{n_1}}{\sqrt{\frac{(\frac{s_0+s_1}{n_0+n_1})(1-\frac{s_0+s_1}{n_0+n_1})(n_0+n_1)}{n_0 n_1}}} \text{ is either } \begin{cases} \leq -z_{\beta/2} \\ \text{or} \\ \geq +z_{\beta/2} \end{cases},$$

where $z_{\beta/2}$ is a standard normal deviate (Larsen & Marx, 1986). Because sample size is important for this test, the algorithm is slightly modified to ensure that, at the beginning of a run, each action is chosen a minimum number of times. This parameter will be referred to as β_{min}.

The complexity of this algorithm is the same as that of the LARCKDNF algorithm of section 5.1, namely $O(M^k)$.

6. Experimental Results

This section reports the results of a set of experiments designed to compare the performance of the algorithms discussed in this paper with one another on a set of problems that illuminates their relative strengths and weaknesses.

6.1. Algorithms and Tasks

The following algorithms were tested in these experiments:

- IE (Defined in figure 1)
- LARC (Defined in figure 2)
- BPRC (Defined in figure 3)
- LARCKDNF (Defined in figure 4)
- IEKDNF (Defined in figure 6)

The regular interval estimation algorithm IE is included as a yardstick; it is computationally much more complex than the other algorithms and may be expected to out-perform them.

Each of the algorithms was tested in three different tasks. The tasks are called *binomial Boolean expression worlds* and can be characterized by the parameters M, *expr*, p_{1s}, p_{1n}, p_{0s}, and p_{0n}. The parameter M is the number of input bits; *expr* is a Boolean expression over the input bits; p_{1s} is the probability of receiving reinforcement value 1 given that action 1 is taken when the input instance satisfies *expr*; p_{1n} is the probability of receiving reinforcement value 1 given that action 1 is taken when the input instance does not satisfy *expr*; p_{0s} is the probability of receiving reinforcement value 1 given that action 0 is taken when the input instance satisfies *expr*; p_{0n} is the probability of receiving reinforcement value 1 given that action 0 is taken when the input instance does not satisfy *expr*. Input vectors are chosen randomly by the world according to a uniform probability distribution.

Table 1. Parameters of tasks for k-DNF experiments.

Task	M	*expr*	p_{1s}	p_{1n}	p_{0s}	p_{0n}
1	3	$(i_0 \wedge i_1) \vee (i_1 \wedge i_2)$	.6	.4	.4	.6
2	3	$(i_0 \wedge \neg i_1) \vee (i_1 \wedge \neg i_2) \vee (i_2 \wedge \neg i_0)$	.9	.1	.1	.9
3	10	$i_2 \wedge \neg i_5$	.9	.6	.5	.8

Table 2. Best parameter values for each algorithm on each task.

ALG	param	task 1	task 2	task 3
LARCKDNF	α	.125	.25	.001
IEKDNF	$z_{\alpha/2}$	3	3.5	2.5
	$z_{\beta/2}$	1	2.5	3.5
	β_{min}	15	5	30
LARC	α	.125	.0625	.03
BPRC	β	.1	.25	.1
	β_h	.2	.3	.1
	ρ	.15	.15	.3
	ρ_h	.2	.05	.3
IE	$z_{\alpha/2}$	3.0	1.5	2.0

Table 1 shows the values of these parameters for each task. The first task has a simple, linearly separable function; what makes it difficult is the small separation between the reinforcement probabilities. Task 2 has highly differentiated reinforcement probabilities, but the function to be learned is a complex exclusive-or. Finally, task 3 is a simple conjunctive function, but all of the reinforcement probabilities are high and it has significantly more input bits than the other two tasks.

6.2. Parameter Tuning

Each of the algorithms has a set of parameters. For both IEKDNF and LARCKDNF, k is set to 2. Algorithms LARC and LARCKDNF have parameters α, β, and σ. Following Sutton (1984), parameters β and σ in LARCKDNF and LARC are fixed to have values .1 and .3, respectively.[4] The IEKDNF algorithm has two confidence-interval parameters, $z_{\alpha/2}$ and $z_{\beta/2}$, and a minimum age for the equality test β_{min}, while the IE algorithm has only $z_{\alpha/2}$. The BPRC algorithm has a large set of parameters: β, learning rate of the evaluation output units, β_h, learning rate of the evaluation hidden units, ρ, learning rate of the action output units, and ρ_h, learning rate of the action hidden units. All of the parameters for each algorithm are chosen to optimize the behavior of that algorithm on the individual task. The success of an algorithm is measured by the average reinforcement received per tick, averaged over the entire run.

Table 3. Average reinforcement for tasks over 100 runs of length 3000.

ALG-TASK	1	2	3
IE	.5827	.8966	.7205
LARC	.5456	.7459	.7644
BPRC	.5456	.7406	.7620
LARCKDNF	.5783	.8903	.7474
IEKDNF	.5789	.8900	.7939
random	.5000	.5000	.7000
optimal	.6000	.9000	.8250

For each algorithm and task, a series of 100 trials of length 3000 were run with different parameter values. Table 2 shows the best set of parameter values found for each algorithm-task pair.

6.3. Results

Using the best parameter values for each algorithm and task, the performance of the algorithms was compared on runs of length 3000. The performance metric was average reinforcement per tick, averaged over the entire run. The results are shown in table 3, together with the expected reinforcement of executing a completely random behavior (choosing actions 0 and 1 with equal probability) and of executing the optimal behavior.

These results do not tell the entire story, however. It is important to test for statistical significance to be relatively sure that the ordering of one algorithm over another did not arise by chance. Figure 7 shows, for each task, a pictorial representation of the results of a 1-sided t-test applied to each pair of experimental results. The graphs encode a partial order of statistically significant dominance, with solid lines representing significance at the .95 level.

With the best parameter values for each algorithm, it is also instructive to compare the rate at which performance improves as a function of the number of training instances. Figures 8, 9, and 10 show superimposed plots of the learning curves for each of the algorithms. Each point represents the average reinforcement received over a sequence of 100 steps, averaged over 100 runs of length 3000. Note that in order to illuminate the relative performance of the algorithms, the graphs have different horizontal scales, with the performance of the random algorithm as the baseline.

6.4. Discussion

First, it is important to note that the comparison of these algorithms is not strictly fair. The LARC and BPRC algorithms were designed to work in environments with real-valued inputs and reinforcement signals and perhaps on much larger spaces. In addition, their biases and generalization characteristics are quite different from those of the k-DNF algorithms.

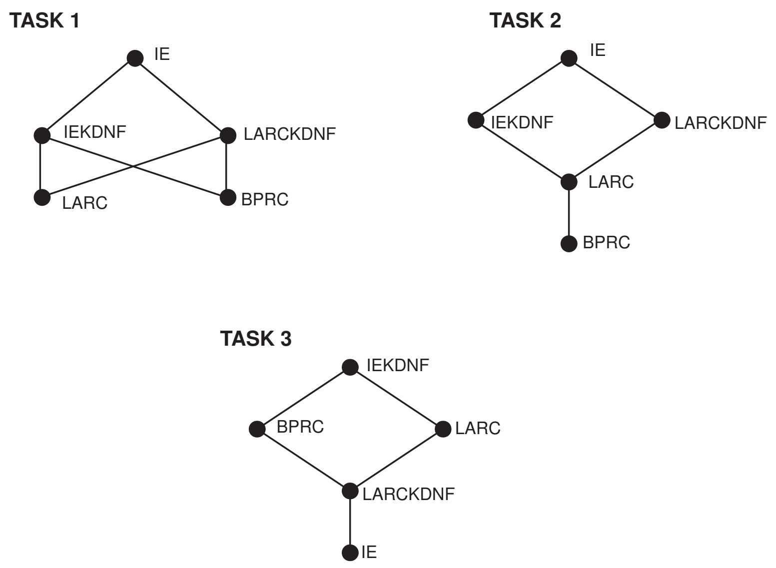

Figure 7. Significant dominance partial order among algorithms for each task.

They are included as benchmarks for the k-DNF algorithms because there are no other appropriate algorithms for comparison.

On tasks 1 and 2, the basic, table-driven, interval estimation algorithm, IE, performed significantly better than any of the other algorithms. The magnitude of its superiority, however, is not extremely great—figures 8 and 9 reveal that the IEKDNF and LARCKDNF algorithms have similar performance characteristics both to each other and to IE. On these two tasks, the overall performance of IEKDNF and LARCKDNF were not found to be significantly different.

The backpropagation algorithm, BPRC, performed considerably worse than expected on tasks 1 and 2. It is very difficult to tune the parameters for this algorithm, so its poor performance may be explained by a sub-optimal setting of parameters.[5] However, it is possible to see in the learning curves of figures 8 and 9 that the performance of BP was still increasing at the ends of the runs. This may indicate that with more training instances it would eventually converge to optimal performance.

The LARC algorithm performed poorly on both tasks 1 and 2. This poor performance was expected on task 2, because linear associators are known to be unable to learn non-linearly-separable functions (Minsky & Papert, 1969). Task 2 is difficult for LARC because, during the execution of the algorithm, the evaluation function can be too complex to be learned by the simple linear associator, even though the action function is linearly separable.

Task 3 reveals many interesting strengths and weaknesses of the algorithms. One of the most interesting is that IE suddenly becomes the worst performer. Because the target function is simple and there is a larger number of input bits, the ability to generalize across

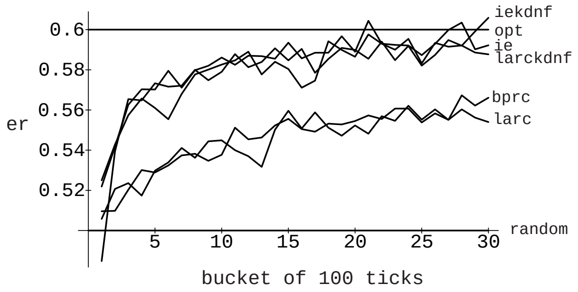

Figure 8. Learning curves for Task 1.

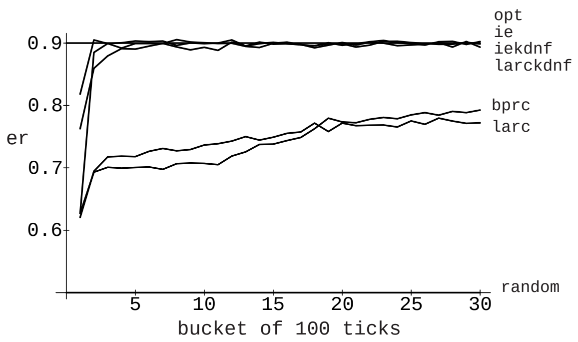

Figure 9. Learning curves for Task 2.

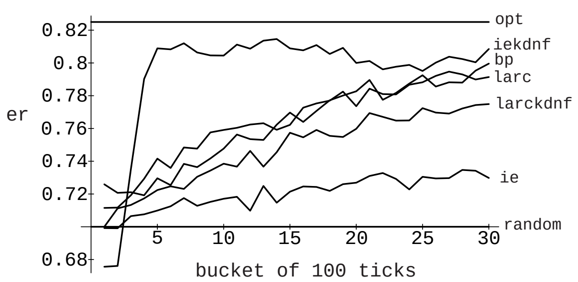

Figure 10. Learning curves for Task 3.

input instances becomes crucial. The IEKDNF algorithm is able to find the correct action function early during the run (this is apparent in the learning curve of figure 10). However, because the reinforcement values are not highly differentiated and because the size of the set T is quite large, it begins to include extraneous terms due to statistical fluctuations in the environment, causing slightly degraded performance. The BPRC and LARCKDNF algorithms have very similar performance on task 3, with the LARC algorithm performing slightly worse, but still reasonably well. The good performance of the generalizing algorithms is especially apparent when we consider the size of the input space for this task. With 10 input bits, by the end of a run of length 3000, each input can only be expected to have been seen about 3 times. This accounts for the poor performance of IE, which would eventually reach optimal asymptotic performance on longer runs.

7. Relaxing the Assumptions

Now we can consider the consequences of relaxing the assumptions made at the beginning of this paper. In some cases, simple changes can be made to the algorithms that will allow them to work in the more general situations. In others, there are theoretical problems that make extensions difficult. Each of the concrete extensions proposed has been implemented and tested.

Thus far we have assumed that the agent has only two possible actions. Many of the early learning-automata algorithms are directly applicable to problems with more than two actions. It has also been shown (Kaelbling, 1993b) that the problem of generating actions specified by N output bits can be solved by N interconnected modules, each of which learns to generate one output bit from reinforcement. Thus, the algorithms presented here could be applied, using this method, to problems with many possible outputs.

The problem of delayed reinforcement has been addressed by Sutton (1988) and Watkins (1989), among others. Sutton's solution, called the *temporal difference method* (TD), can be abstracted away from the particular reinforcement-learning mechanism being used. It provides a module that learns to transduce the delayed reinforcement signal that is coming from the world into an immediate reinforcement signal that evaluates each state of the world to be the expected future reward based on the agent's current strategy. Because this local reinforcement signal must be learned, using a TD module violates a different one of our assumptions: that the expected reinforcement of performing an action in a situation be fixed over the course of a run. This will be addressed below.

If the reinforcement values are not Boolean, but the trials are independent, we have a variety of statistical models available. The LARCKDNF algorithm, as presented, can be used when the reinforcement is real-valued. The IEKDNF algorithm can be implemented with different statistical tests. For instance, if we know that the reinforcement values for each input-action pair are normally distributed, we can use standard statistical methods to construct confidence intervals and to test for equality of means. If we have no *a priori* model of the distribution of reinforcement, we can use the methods of non-parametric statistics.

All of the algorithms we have considered use some method of estimating the center of the distribution of reinforcement values; the only further thing needed for the interval estimation algorithms is a method for estimating the spread of the distribution. If the distribution of reinforcements is known, then the standard statistical estimators are provably good. When reinforcements are Boolean and independent, they can always be modeled, as in the body of this paper, with a Bernoulli distribution. It often be the case, though, that the distribution is not known. In such cases, we have two alternatives: use a parametric model that is probably wrong or use a non-parametric model. The mean and the median are both good estimators for the centers of distributions; the difficulty usually lies in deciding how to estimate the spread. I have found that standard non-parametric techniques (Gibbons, 1985) work reasonably well, but can be computationally expensive; careful application of the techniques for normal distributions can also be effective, even when the distribution is non-normal.

If the individual reinforcement values are not independent given the input and action, then the statistical methods are no longer theoretically grounded. They can still be considered as well-motivated heuristic methods, to be compared empirically to those in the neural-network based approaches, for instance.

Finally, we consider the case of having the expected reinforcement of performing an action in a situation change during the course of a run. The LARCKDNF algorithm will work in such cases, although it might be necessary to adjust its parameters. The statistically-based IEKDNF and IE algorithms can be modified to work, by causing their statistics to decay over time. If an action has not been tried for a long time, its n value will slowly decay, which will cause its confidence interval to grow larger. Eventually it will grow large enough for that action to be chosen again. If the action has good results, the policy will be changed to favor this action.

8. Conclusion

From this study, we can see that it is useful to design algorithms that are tailored to learning certain restricted classes of functions. On tasks drawn from the appropriate class, the specially-designed algorithms presented here out-performed standard methods of comparable complexity; the methods based on overt statistical tests, IE and IEKDNF, converged to good strategies much more quickly than the algorithms based on artificial neural-network techniques. In addition, the statistical algorithms have internal semantics that are clear and directly interpretable in the language of classical statistics. This simplifies the process of extending the algorithms to apply to other types of tasks in a principled manner.

Important future work will be to identify other restricted classes of functions that can be learned efficiently and effectively from reinforcement and demonstrate that these classes contain functions that solve interesting and important problems from the real world.

Acknowledgements

Thanks to Stan Rosenschein for providing financial and moral support and to Rich Sutton for helpful discussions of neural-network and statistical reinforcement learning methods. An anonymous reviewer supplied a number of useful comments and questions.

This work was supported in part by the Air Force Office of Scientific Research under contract F49620-89-C-055, in part by the System Development Foundation, in part by Teleos Research IR&D, in part by National Science Foundation National Young Investigator Award IRI-9257592 and in part by ONR Contract N00014-91-4052, ARPA Order 8225.

Notes

1. This is a somewhat more complex form than usual, designed to give good results for small values of n (Larsen & Marx, 1986).
2. The choice of L is not relevant to our reinforcement-learning scenario—the details are described by Valiant (1984; 1985).
3. Valiant's presentation of the algorithm defines T to be the set of conjunctive terms of length k *or less* over the set of atoms and their negations; however, because any term of length less than k can be represented as a disjunction of terms of length k, we use a smaller set T for simplicity in exposition and slightly more efficient computation time. This simplification will result, in the *iekdnf* algorithm, in a decreased tendency to generalize (good results for the term $i_0 \wedge i_1$ would also be compiled in the statistics for term i_0, making it likely to generate a 1 given the novel example satisfying $i_0 \wedge \neg i_1$). It also results in the reduction of false positive results by simply eliminating the number of terms that can generate them without reducing the representational power.
4. This strategy seemed to work well until LARCKDNF was applied to task 3. In this situation, there are 180 inputs to the linear associator; with so many inputs, the large value of β causes the weights to grow without bound. To remedy this problem, but to avoid more parameter tuning, for task 3, β was set to the same value as α.
5. In the parameter tuning phase, the parameters were varied independently—it may well be necessary to perform gradient-ascent search in the parameter space, but that is a computationally difficult task, especially when the evaluation of any point in parameter space may have a high degree of noise.

References

Anderson, C. W. (1986). *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Amherst, Massachusetts.

Barto, A. G., Sutton, R. S., & Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40, 201–211.

Berry, D. A. & Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. London: Chapman and Hall.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. New York, New York: Academic Press.

Gibbons, J. D. (1985). *Nonparametric Statistical Inference*. New York and Basel: Marcel Dekker, Inc.

Gluck, M. A. (1991). Stimulus generalization and representation in adaptive network models of category learning. *Psychological Science*, 1(1), 50–55.

Hertz, J., Krogh, A., & Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Redwood City, California: Addison Wesley.

Kaelbling, L. P. (1993a). Associative reinforcement learning: A generate and test algorithm. *Machine Learning*. To appear.

Kaelbling, L. P. (1993b). *Learning in Embedded Systems*. Cambridge, Massachusetts: The MIT Press. Also available as a PhD Thesis from Stanford University, 1990.

Larsen, R. J. & Marx, M. L. (1986). *An Introduction to Mathematical Statistics and Its Applications*. Englewood Cliffs, New Jersey: Prentice-Hall.

Minsky, M. L. & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, Massachusetts: The MIT Press.

Munro, P. (1987). A dual back-propagation scheme for scalar reward learning. In *Proceedings of the Ninth Conference of the Cognitive Science Society* (pp. 165–176). Seattle, Washington.

Narendra, K. & Thathachar, M. A. L. (1989). *Learning Automata: An Introduction*. Englewood Cliffs, New Jersey: Prentice-Hall.

Rosenblatt, F. (1961). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington, DC: Spartan Press.

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, Massachusetts.

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

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.

Valiant, L. G. (1985). Learning disjunctions of conjunctions. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1 (pp. 560–566). Los Angeles, California: Morgan Kaufmann.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 339–356.

Widrow, B. & Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record* New York, New York. Reprinted in *Neurocomputing: Foundations of Research*, James A. Anderson and Edward Rosenfeld, editors, The MIT Press, Cambridge, Massachusetts, 1988.

Williams, R. J. (1988). On the use of backpropagation in associative reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks* San Diego, California.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3), 229–256.

Wolpert, D. H. (1993). *On Overfitting Avoidance as Bias*. Technical Report 93-03-016, Santa Fe Institute, Santa Fe, New Mexico.