Function Optimization Using Connectionist Reinforcement Learning Algorithms*

Ronald J. Williams and Jing Peng
College of Computer Science
Northeastern University

Appears in *Connection Science, 3*, pp. 241-268, 1991.

Abstract

Any nonassociative reinforcement learning algorithm can be viewed as a method for performing function optimization through (possibly noise-corrupted) sampling of function values. We describe the results of simulations in which the optima of several deterministic functions studied by Ackley (1987) were sought using variants of REINFORCE algorithms (Williams, 1987; 1988). Some of the algorithms used here incorporated additional heuristic features resembling certain aspects of some of the algorithms used in Ackley's studies. Differing levels of performance were achieved by the various algorithms investigated, but a number of them performed at a level comparable to the best found in Ackley's studies on a number of the tasks, in spite of their simplicity. One of these variants, called REINFORCE/MENT, represents a novel but principled approach to reinforcement learning in nontrivial networks which incorporates an entropy maximization strategy. This was found to perform especially well on more hierarchically organized tasks.

1 Introduction

1.1 Background

In his thesis, Ackley (1987) explored an interesting general approach to function optimization differing somewhat from more common approaches which emphasize limiting behaviors or terminating conditions. He studied the behavior of a variety of general-purpose optimization algorithms, some already existing and some of his own design, on a number of optimization problems involving functions defined on binary n-tuples, and measured their performance using a criterion which does not require convergence. The algorithms he investigated were: 1) two forms of simple hillclimbing algorithm, combined with random restarts following convergence; 2) simulated annealing (Kirkpatrich, Gelatt, & Vecchi, 1983), combined with random restarts following convergence;

*Preparation of this paper was partially supported by Grant IRI-8921275 from the National Science Foundation.

3) a fixed-temperature "thermally agitated" hillclimber (essentially simulated annealing but without the annealing); 4) two variations on genetic algorithms (Goldberg & Holland, 1988; Holland, 1975), combined with random restarts following convergence; 5) a combination of hillclimbing and genetic search, together with random restarts following convergence; and 6) a connectionist network algorithm called *stochastic iterated genetic hillclimbing* (SIGH), which contains elements drawn from a number of sources and has an interesting description in terms of an election metaphor. Not studied were a number of existing reinforcement learning algorithms which are also appropriate candidates, such as various algorithms from the stochastic learning automata literature (Narendra & Thathatchar, 1989), and various connectionist reinforcement learning algorithms, such as the *associative reward-penalty* algorithm of Barto and colleagues (Barto, 1985; Barto & Anandan, 1985; Barto & Anderson, 1985) and REINFORCE algorithms (Williams, 1987; 1988). In this paper we describe the results of experiments performed using several variants of REINFORCE on some of the same tasks studied by Ackley. A preliminary report of this work appeared in (Williams & Peng, 1989).

1.2 The Problem Formulation

The problem formulation proposed by Ackley and adopted by us here can be loosely described as follows. Consider a generate-and-test scenario in which there is no particular stopping criterion. The job of the generator is to generate trial points and the tester determines the value of the function to be optimized at the given trial point. Because there is no stopping criterion, this process of generate-and-test is repeated indefinitely. To make it interesting, however, we would like the generator to be adaptive, so that it tends to generate better points as it gains information about the function being optimized. We adopt the convention that a "better" point is one having a higher function value.

The systems of interest are thus ones which perform an endless repetition of a simple generate-test-adapt loop. To view such a system as carrying out an optimization algorithm, one has to add a stopping criterion of some sort, or at least a way of reporting out some answer during any step of the process. There are several ways these issues might be handled. One way, suitable for a process which eventually converges, so that eventually the same trial point is always generated, is to test for this condition and report this point as the output of the algorithm. Another way is to always retain a memory of the best point found so far and report this point as the output of the algorithm whenever it is to be terminated. The real value of this approach, however, is its attempt to isolate as much as possible the useful features of the generate-test-adapt loop from issues involving the choice of termination criterion. In particular, this approach does not require that a global optimum be recognized as such, either explicitly, so that the process actually terminates when a global optimum is generated, or implicitly, with eventual convergence to the global optimum. The generator may continue generating inferior points even after having generated a global optimum.

Thus this approach stresses what Ackley has called *sustained exploration*. We believe

that sustained exploration may turn out to be an important feature of procedures designed to perform some sort of optimization process,[1] particularly when very large (yet finite) search spaces (e.g., having 2^{1000} points) are involved, or when the function to be optimized actually varies over time (both of which one might expect to be characteristic of many realistic optimization problems).

Of course, to study various approaches to the adaptive generation process through simulation, it is necessary to incorporate the adaptive generators to be studied into terminating algorithms. We follow Ackley and adopt the strategy of using problems for which an optimum point is known and simply run the algorithm until either such a point is first generated or some maximum computational effort has been expended. The measure of computational effort used is the number of function evaluations performed. This measure is used both to determine when to terminate a run (and declare failure) and as a measure of the performance of the various adaptive generation techniques studied.

1.3 Reinforcement Learning Networks as Adaptive Trial Generators

A connectionist network can be used as a generator of trial points for any optimization problem as long as the domain of the function to be optimized can be represented as the set of output patterns of the network. Furthermore, any reinforcement learning algorithm for adjusting the weights in the network (e.g., Barto, 1985; Barto & Anandan, 1985; Barto & Anderson, 1985; Munro, 1987; Sutton, 1984; Williams, 1986; 1987; 1988) can be used to provide a means of adapting the behavior of this trial point generator simply by regarding the function value as the reinforcement signal delivered to the network in response to its output pattern.

This particular use of a reinforcement learning algorithm actually represents an extreme specialization of a much more general formulation of the reinforcement learning problem in which: 1) the reinforcement may be a stochastic rather than deterministic function of network output; 2) the network may be required to perform a mapping from input to output (*associative reinforcement learning*); and 3) the reinforcement signal and input provided to the network may depend on past input and/or output patterns (so that the environment has memory). This more general formulation encompasses features much more characteristic of realistic learning situations. For this reason, there are a number of sophisticated directions in which the study of such reinforcement learning systems has been taken, including the use of temporal difference learning methods for adaptive prediction of future reinforcement (Barto, Sutton, & Anderson, 1983; Sutton, 1984; Sutton, 1988a) and the use of internal models of the environment of one type or other (Munro, 1987; Sutton & Pinette, 1985), but most of these are not relevant to our purposes here. In this paper we treat the problem of maximizing a deterministic function as an extremely pared-down form of reinforcement learning task, one which is nonassociative and involves a memoryless environment and noise-free reinforcement

[1] The term *optimization process* here is meant to cover a variety of possibilities, including processes which actually terminate at a global maximum and processes which simply continue to discover better and better points whenever possible but do not terminate.

signal. An important feature of the type of problem we seek to attack with these techniques is the large space over which the optimization is to be performed. This differs markedly from the emphasis in 2-armed bandit and similar problems heavily studied in the learning automata literature (Narendra & Thathatchar, 1989). While these are also reinforcement learning tasks, the issue in such problems is how to sample the entire space enough to determine with statistical confidence which action gives the best average result while also converging toward selecting that action exclusively. Here, however, we consider search spaces which are so large that it is infeasible to consider sampling more than a miniscule subset of all the possible points.

We also emphasize that the connectionist approach to optimization we study here is quite different from that initiated by Hopfield and Tank (1985). In that approach, much more knowledge of the function being optimized is used than in the approach taken here. That approach requires devising a function having the same global optimum as the actual function to be optimized. This new function must be one which can be incorporated into the network's weights so that the network always settles into local optima of this function. In contrast, the technique being studied here makes use of no general information concerning the function being optimized. Instead, the only information that becomes available about the function arises through sampling function values at various points. This is the same paradigm studied by those exploring the use of genetic algorithms (Goldberg & Holland, 1988; Holland, 1975) for function optimization, where the function to be optimized is usually called the *fitness* function, in accord with an approach inspired by the process of biological evolution through natural selection.

As in most approaches to reinforcement learning, the networks we study here use stochastic units to allow sampling of a variety of output patterns. Thus for any given setting of the weights in the network, a particular distribution of output patterns is generated. The broad goal of any adaptive sampling scheme is to try to reshape the sampling distribution in ways that make it more likely to sample better points. Applying a reinforcement learning algorithm to the weights can be viewed as a means of doing just this when the sample points are generated by a network.

2 Formal Notation and Terminology

Before discussing the specific algorithms used and the optimization problems investigated, we first introduce the general notational framework and mathematical assumptions used in the description of reinforcement-learning networks throughout this paper.

2.1 Network Quantities

Consider a network having n_I external input lines from the environment, n_O output units which affect the environment, and n_H hidden units. We let $\mathbf{x}^{\mathcal{N}}$ denote the n_I-tuple of external input signals to the network at a particular time and we let $\mathbf{y}^{\mathcal{N}}$ denote the n_O-tuple of network output produced as a result. It is also convenient to collect all the unit output values into the $(n_O + n_H)$-tuple $\mathbf{y}$. A typical element of $\mathbf{y}$ is y_i, the output

of the i^{th} unit in the network. In addition, we define $\mathbf{x}$ to be the $(n_I + n_H + n_O)$-tuple obtained by concatenating $\mathbf{x}^{\mathcal{N}}$ with $\mathbf{y}$. Let U denote the set of indices used to designate units in the networks and I the disjoint set of indices used to designate input lines. Then a typical element of $\mathbf{x}$ is x_j, which is either the output of the j^{th} unit in the network, if $j \in U$, or the value received on the j^{th} input line, if $j \in I$.

Note that one consequence of our notational convention is that x_k and y_k are two different names for the same quantity when $k \in U$. The general philosophy behind our use of this notation is that variables symbolized by x represent input and variables symbolized by y represent output, whether at the level of individual units or at the level of the entire network. Since the output of a unit may also serve as input to other units, we will consistently use x_k when its role as input is being emphasized and y_k when its role as output is being emphasized. Similarly, the input to the network is $\mathbf{x}^{\mathcal{N}}$ and its output is $\mathbf{y}^{\mathcal{N}}$.

Let $\mathbf{W}$ denote the weight matrix for the network, with exactly one weight (which may be zero) between each pair of units and also from each input line to each unit. The element w_{ij} of this $(n_H + n_O) \times (n_I + n_H + n_O)$ matrix represents the weight on the connection to the i^{th} unit from either the j^{th} unit or the j^{th} input line. To accommodate a bias weight for each unit, we simply include among the n_I input lines one input whose value is always 1. We adopt the convention that this bias input has an index of 0, so that w_{i0} represents the bias weight for the i^{th} unit.

Here we have introduced notation appropriate for the general case when the network may be provided with external environmental input. Thus this includes the associative case. For the specific function optimization application considered here, we assume that there is no external input (except that we still need the bias input). Each time the network computes an output vector $\mathbf{y}^{\mathcal{N}}$, this is assumed to represent a single trial point for the function to be optimized.

2.2 Bernoulli Logistic Units

In this paper we assume that the units in the network are *Bernoulli logistic units*, in the terminology of (Williams, 1986; 1987; 1988). These are appropriate for our purposes because the functions to be optimized are defined over binary n-tuples. The output y_i of such a unit is either 0 or 1, determined stochastically using the Bernoulli distribution with parameter

$$p_i = f(s_i), \tag{1}$$

where f is the logistic function $f(s_i) = 1/(1 + e^{-s_i})$ and

$$s_i = \sum_{j \in U \cup I} w_{ij} x_j \tag{2}$$

is the usual weighted summation of input values to that unit. For such a unit, p_i represents its probability of choosing 1 as its output value.

2.3 REINFORCE Algorithms

The main objective of the research reported here was to study how well the REINFORCE class of algorithms would perform when used in function optimization tasks. Here we give a brief overview of the particular form such algorithms take when applied to the networks of Bernoulli logistic units that we use in this paper. For extensive discussion of these algorithms, see Williams (1988).

In the general reinforcement learning paradigm, the network generates output pattern $\mathbf{y}^{\mathcal{N}}$ and the environment responds by providing the reinforcement r as its evaluation of that output pattern, which is then used to drive the weight changes according to the particular reinforcement learning algorithm being used by the network. For the Bernoulli logistic units used here, a general REINFORCE algorithm prescribes weight increments equal to

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij})(y_i - p_i)x_j, \tag{3}$$

where α_{ij} is a positive learning rate (possibly different for each weight) and b_{ij} serves as a *reinforcement baseline* (which can also be different for each weight). Here we consider only algorithms having the form

$$\Delta w_{ij} = \alpha(r - b)(y_i - p_i)x_j, \tag{4}$$

where $\alpha_{ij} = \alpha$ and $b_{ij} = b$ for all i and j. It can be shown (Williams, 1986; 1988) that, regardless of how b is computed, whenever it does not depend on the immediately received reinforcement value r, such an algorithm satisfies

$$E\{\Delta \mathbf{W} | \mathbf{W}\} = \alpha \nabla_{\mathbf{W}} E\{r | \mathbf{W}\}, \tag{5}$$

where E denotes the expectation operator. A reinforcement learning algorithm satisfying (5) can be loosely described as having the property that it statistically climbs the gradient of expected reinforcement.

3 The Optimization Problems

The optimization problems whose simulation results we report here are particular maximization problems originally studied by Ackley (1987). Of the six problems we detail here, four are contrived problems designed to isolate specific features that various optimization problems may possess, and the other two are specific combinatorial optimization problems. We make no claim that this suite of problems represents a definitive benchmark for optimization algorithms; we simply wished to be able to compare our results with Ackley's.

Throughout this paper we use J generally to denote a function to be maximized, with $\mathbf{u}$ representing a point in its domain. For each of the problems studied here, J is a mapping from the n-dimensional hypercube $\{0, 1\}^n$ into the real numbers, so each point

$\mathbf{u}$ its domain is an n-dimensional bit vector $(u_1, u_2, \ldots, u_n)$. Other notation we use in the description of these problems is as follows:

$$\begin{aligned} n_1 &= \sum_i u_i = \text{the number of 1s in } \mathbf{u}; \\ n_0 &= n - n_1 = \text{the number of 0s in } \mathbf{u}; \\ \mathbf{0} &= \text{the } n\text{-tuple } \mathbf{u} \text{ such that } u_i = 0 \text{ for all } i; \text{ and} \\ \mathbf{1} &= \text{the } n\text{-tuple } \mathbf{u} \text{ such that } u_i = 1 \text{ for all } i. \end{aligned}$$

3.1 Abstract Problems

The *One-Max* function is given by

$$J(\mathbf{u}) = 10n_1, \tag{6}$$

which has the point $\mathbf{1}$ as its global maximum and no false maxima.

The *Two-Max* function is given by

$$J(\mathbf{u}) = |18n_1 - 8n|, \tag{7}$$

which has a global maximum at $\mathbf{1}$ and a false maximum at $\mathbf{0}$. The function value is $10n$ at the global maximum (as it is for the other 3 abstract problems as well) and $8n$ at the false maximum. The number of points in the space for which uphill moves lead to the global maximum is somewhat larger than the number for which the false maximum might look attractive to a hillclimber.

The *Porcupine* function is given by

$$J(\mathbf{u}) = 10n_1 - 15(n_0 \bmod 2). \tag{8}$$

This is essentially the same as One-Max, but with a "high-frequency" component added on to confound any myopic hillclimber. The global maximum is at $\mathbf{1}$, but every point whose Hamming distance from $\mathbf{1}$ is even is a local maximum.

The *Plateaus* function is defined as follows, where it is assumed that n is divisible by 4:

$$J(\mathbf{u}) = \sum_{k=1}^{4} J_k(\mathbf{u}), \tag{9}$$

where

$$J_k(\mathbf{u}) = 2.5n \prod_{i=kn/4+1}^{(k+1)n/4} u_i. \tag{10}$$

In words, this function is computed as follows: Divide the bits into four equal-sized groups. For each group compute a score which is $2.5n$ if all the bits in that group are 1 and is 0 otherwise. Then $J(\mathbf{u})$ is the sum of these four scores. Like the previous ones, this function has a global maximum at $\mathbf{1}$. It also has very large plateaus over which the function is constant.

For all of these abstract optimization problems, we report the results of simulation studies for the case where $n = 20$.

3.2 Graph Partitioning Problems

Another type of problem we have investigated is based on a type of combinatorial optimization problem known to be NP-complete, the minimum-cut graph partitioning problem (Garey & Johnson, 1979). The usual form of this problem is: Given a graph with an even number of nodes, assign each node to one of two groups in such a way that the two groups contain equal numbers of nodes and the number of edges connecting nodes lying in both groups is minimized. If there are n nodes in the graph, any assignment of nodes to groups can be represented by giving the groups the names 0 and 1 and considering a binary n-tuple $\mathbf{u}$ to represent the assignment in which the i^{th} node is assigned to the group u_i. In order to treat this as an optimization problem over the set of all binary n-tuples, the hard constraint that the two groups contain equal numbers of nodes must be replaced by an imbalance penalty. Thus, following Ackley, we treated the graph partitioning problem as the problem of maximizing the function

$$J(\mathbf{u}) = -c(\mathbf{u}) - 0.1(n_1 - n_0)^2, \tag{11}$$

where $c(\mathbf{u})$ is the number of edges which cross the partition for the particular assignment $\mathbf{u}$.

Ackley studied both randomly generated graphs and graphs having a particular hierarchical structure. In the belief that it is for those problems having some type of higher-order structure that one might hope to develop useful general-purpose optimization algorithms, we restricted our attention to the two hierarchically structured graphs he studied. One such graph has 32 nodes and is depicted in Figure 1, while the other has 64 nodes and is depicted in Figure 2. Ackley called these *multilevel hypercube graphs* and gave them the names *MLC-32* and *MLC-64*, respectively, which we also use here.

Insert Figure 1 about here.

Insert Figure 2 about here.

Note that the function to be optimized in such problems is symmetric under bitwise complementation of its argument for any graph. For these particular graph partitioning problems there are exactly two global maxima, but a number of local maxima. Furthermore, the minimum Hamming distance between any false peak and a global maximum is 8 for both MLC-32 and MLC-64.

Note that when a network is used to generate the trial points $\mathbf{u}$, the number of output units n_O in the network must obviously be equal to n, and we identify the network output pattern $\mathbf{y}^{\mathcal{N}}$ with $\mathbf{u}$.

4 Experiments Using Team Networks

4.1 Team Networks

Here we report the results of one set of studies using a very simple form of trial generating network in which all of the units are output units and there are no interconnections between them. Later in the paper we discuss experiments using networks where interconnections play an important role. This degenerate class of network corresponds to what is called a *team* of automata in the literature on stochastic learning automata (Narendra & Thathatchar, 1989). We thus call these networks *teams of Bernoulli logistic units.* Because all units are output units, we can use $\mathbf{y}^{\mathcal{N}}$ and $\mathbf{y}$ interchangeably for such a network. Figure 3 shows an example of a team network.

Insert Figure 3 about here.

For any Bernoulli logistic unit receiving no input from any sources external to that unit except the constant bias input, the probability that that unit produces a 1 on any particular trial given the value of its bias weight w_{i0} is

$$Pr\{y_i = 1 | w_{i0}\} = p_i = f(s_i) = \frac{1}{1 + e^{-w_{i0}}}. \tag{12}$$

Because all units pick their outputs independently, it follows that for such a team of Bernoulli logistic units the probability of any particular output vector conditioned on the current value of the weight matrix $\mathbf{W}$ is given by

$$Pr\left\{\mathbf{y}^{\mathcal{N}} | \mathbf{W}\right\} = \prod_{i \in U} p_i^{y_i} (1 - p_i)^{1 - y_i}. \tag{13}$$

The bias weights w_{i0} are adjusted according to the particular learning algorithm used, and the details of these algorithms are discussed below. Here we note that when $w_{i0} = 0$, the unit is equally likely to pick either 0 or 1, while increasing w_{i0} makes a 1 more likely. Adjusting the bias weights in a team of Bernoulli logistic units is thus tantamount to adjusting the probabilities for the individual components.

4.2 Team Algorithms Used

We used a total of six different algorithms with the team architecture, which are all of the same general form: At the t^{th} time step, after generating output $\mathbf{y}^{\mathcal{N}}(t)$ and receiving reinforcement $r(t) = J(\mathbf{y}(t))$, increment each bias weight w_{i0} by

$$\Delta w_{i0}(t) = \alpha \rho(t) e_{i0}(t) - \delta w_{i0}(t), \tag{14}$$

where α, the learning rate, and δ, the weight decay rate, are parameters of the algorithm. We call ρ the *reinforcement factor* and e_{i0} the *eligibility* of the weight w_{i0}. The

reinforcement factor makes use of an exponentially weighted average, or *trace*, of prior reinforcement values

$$\overline{r}(t) = \gamma\overline{r}(t-1) + (1-\gamma)r(t), \tag{15}$$

and is computed by

$$\rho(t) = r(t) - \overline{r}(t-1) - \beta, \tag{16}$$

where β is a parameter of the algorithm. The trace parameter γ was set equal to 0.9 for all the experiments reported here. Finally, we considered two different forms of eligibility, either

$$e_{i0}(t) = y_i(t) - p_i(t) \tag{17}$$

or

$$e_{i0}(t) = y_i(t) - \overline{y}_i(t-1), \tag{18}$$

where $\overline{y}_i(t)$ is an average of past values of y_i computed by the same exponential weighting scheme used for $\overline{r}$. That is,

$$\overline{y}_i(t) = \gamma\overline{y}_i(t-1) + (1-\gamma)y_i(t). \tag{19}$$

Besides the two forms of eligibility we considered, we varied the algorithms we studied along two other feature dimensions: the size of the decay rate δ and the size of β. In particular, we used two different settings for δ, zero and an appropriate nonzero value, which was chosen to be 0.01 in all our experiments. Similarly, we used two different settings for β, zero and an appropriate nonzero value, which, following Ackley (1987), we chose to be 4. The values 0.01 for the nonzero decay value and 0.9 for the trace parameter were chosen solely because these are commonly used values for parameters that play corresponding roles in a wide variety of similar algorithms; no attempt was made to tune these parameters to explore possible variations in performance. The experimental results we report here used all combinations of these variations except the two where β and δ are both nonzero.

Two of these six algorithms are REINFORCE algorithms, as described earlier, while the other four can be viewed as slight variants of these. The two REINFORCE algorithms are those for which both $e_{i0} = y_i - p_i$ and $\delta = 0$, as can be seen by comparing equation (4) with the result of combining equations (14), (16), and (17). Of these, the one where $\beta = 0$ uses the standard *reinforcement comparison* technique advocated by Sutton (1984), in which the reinforcement baseline b in equation (4) is set equal to $\overline{r}$. This essentially tries to make the reinforcement factor have zero mean. The version where $\beta > 0$ corresponds to setting the reinforcement baseline b in equation (4) equal to $\overline{r} + \beta$. This gives the reinforcement factor a negative bias, which one might expect would destabilize the algorithm whenever r fails to change much. Ackley used this technique as a component of his SIGH algorithm and we were curious to see if this alone could prevent convergence and force sustained exploration.

The use of the $y_i - \overline{y}_i$ form of eligibility was motivated by simulation results of Sutton (personal communication, 1986) suggesting that faster learning could be achieved than with the other form, and there are analytic results suggesting (but not proving) why this

might be so. We omit the details of this analysis, except to remark that the algorithm that results when combining the standard reinforcement-comparison reinforcement factor with this eligibility factor has a close relationship to linear regression analysis when using Bernoulli logistic units.

Finally, the use of weight decay was chosen as a simple heuristic method to force sustained exploration after it was discovered that the other four algorithms always seemed to converge. Having decay on the bias of any particular member of a team of Bernoulli units is very closely related to having a nonzero mutation rate at a particular allele in a genetic algorithm (Goldberg & Holland, 1988; Holland, 1975). To see this, suppose that p_i represents the probability that a particular bit position u_i in a randomly generated bit vector is 1 with no mutation operator present. Then suppose that with probability μ_i a mutation operator complements the bit that would have been generated had there been no mutation. Then the probability that u_i is 1 after this entire process is given by

$$\begin{aligned} Pr\{u_i = 1 | p_i, \mu_i\} &= (1 - \mu_i)p_i + \mu_i(1 - p_i) \\ &= p_i - 2\mu_i(p_i - \frac{1}{2}). \end{aligned}$$

Thus the overall effect of applying a mutation operator in this way is just like applying a proportional decay toward 1/2 of the probability of generating a 1. The only difference between this and proportional bias decay toward 0 is in the nonlinearity of the squashing function. If the squashing function were linear, the two approaches would be identical.

4.3 Results Using Team Algorithms

A summary of the results of the experiments we performed on the six optimization problems, including those using team networks, can be found in Table 1. All parameters were fixed across all problems for each particular algorithm, except for the learning rate α. Some (but not an extensive amount of) fine-tuning was performed on α for each combination of algorithm and problem. Thus the table shows for each such combination the particular value of learning rate used for all the runs of that combination and the rounded mean run length (number of function evaluations) to reach a global optimum for that set of runs. In each case this mean is based on 50 runs of each algorithm/problem combination. Any result reported as infinite indicates that the algorithm failed to find a global maximum in at least some of the runs. As will be discussed below, in every case where this happened the reason was that the algorithm became trapped at a local maximum.

For comparison, we also give the best average result reported by Ackley for each of the optimization problems. His results were reported as rounded means of 50 runs on the first four problems, but the median of 7 runs was used in the graph partitioning problems.

4.4 Discussion of Team Results

These experiments have convinced us that all four team algorithms not having weight decay eventually converge, which we have also found to be true more generally of REINFORCE algorithms in arbitrary networks. We had expected this for the two algorithms having no negative bias in the reinforcement factor, but it turned out to be true even when the negatively biased reinforcement factor was used. This convergence, generally to a false maximum, is the reason why these algorithms were not usually successful in the graph partitioning problems. Thus we conclude that the use of such a negative bias alone is not a mechanism for providing sustained exploration, as we had first believed.

One consequence of this tendency to converge is that the problem for which the choice of learning rate in the algorithms without weight decay was most critical (disregarding the problems on which they typically failed) was the Plateaus function. Too large a learning rate guarantees that the search will converge on one of the plateaus.

One particularly interesting result is that all the algorithms found the maximum of Porcupine virtually as fast as they found the maximum of One-Max. This is almost certainly because these reinforcement learning algorithms are able to handle noisy reinforcement signals and treat the "porcupine quills" as if they were merely noise added to some underlying function (which happens to be One-Max), which they quickly discover and climb.

One general conclusion we draw from our results on the abstract functions is that all of these functions have the property that simple bitwise correlation with the reinforcement function is sufficient to discover the maximum, assuming enough statistics are collected. Clearly the Plateaus function requires waiting the longest for the right statistical pattern to emerge (each 5-bit group must sample the all-1s case at least once), and this is the reason all the algorithms take longer on it.

The results in Table 1 suggest that, for the most part, some speedup is provided by choosing the $y_i - \overline{y}_i$ form of eligibility over that used in REINFORCE. The situations for which the difference is most pronounced are those where the maximum can be found very rapidly. We conjecture that this is due to the fact that the learning algorithms using either form of eligibility are two different estimators of the same underlying quantity and the one using $y_i - \overline{y}_i$ has better small-sample properties.

One result to be noted from Table 1 is that the combination of straight reinforcement comparison with the $y_i - \overline{y}_i$ form of eligibility and weight decay almost always gave the lowest average time to find the maximum among all the team algorithms. While the $y_i - p_i$ form of eligibility gave a better average result in the 64-node graph partioning problem, this difference in performance is not statistically significant since the standard deviation is over 4000 for each of these sets of runs.

The graph partitioning problems obviously provide the real challenge to these algorithms; those with too strong a tendency to hillclimb, even statistically, cannot avoid failing at times by getting stuck on local maxima. One approach we could have adopted to try to make such algorithms work on this problems is the same as that used by Ackley for any algorithm which converges: detect convergence and perform a random

restart. We speculate that perhaps the reason that combining weight decay with these algorithms allows the global maximum to be found is that the refusal to allow weights to grow beyond a certain point amounts to something like a continuing tendency to engage in random restarts coupled with the tendency to climb hills statistically. Indeed, analysis of the runs involving weight decay reveal a behavior which can be loosely described as exploration around a local maximum for a while followed by what may amount to a jump to the neighborhood of another local maximum, exploration of that peak, and so on; eventually such a jump leads to the neighborhood of a global maximum.

5 Experiments With Networks Having Nontrivial Connectivity

5.1 The Importance of Interconnections

The real value of using a learning network to generate trial points for the function to be optimized is that interconnections within the network can enforce coordination of the choices made by the output units in order to concentrate the search in suspected high-payoff regions of the space. This can occur when output units have a direct influence on one another or when hidden units are present which can serve as command cells. For either form of coordination, the presence of randomness in certain parts of the network insures that exploration will occur, but the interconnections insure that this randomness is channelled in appropriate directions. To support this process, the learning algorithm used should allow the network to retain information gained during the exploration process which it can use to control any desired coordination in future trials. With the team approach the only information represented is the result of correlations between the behaviors of individual units and the reinforcement signal. In many problems these first-order correlations carry little or no information about the correct direction to move, but higher-order correlations might be very helpful.

A good example is given by the graph partitioning problems investigated here. Every bit vector representing a partition has the same reinforcement value as its componentwise complement. Thus the first-order correlations of the individual components with the outcome must be zero if we sample uniformly. It is only after some bits become more strongly committed to one value that others discover some correlation between their choice of value and the outcome. A network which can coordinate the choices made by the output units should be able to generate certain combinations of bits with greater probability than if their individual components were selected independently. In particular, for graph partitioning one might hope that units corresponding to a highly interconnected group of nodes in the graph would have their operation coordinated to the point that they would come to select with high probability both the all-0 and the all-1 assignment for that subset of nodes. If the network operates in this way it should expect to find a solution for hierarchical graphs of the type studied here much more quickly than without coordination.

The challenge, of course, is not only to be able to represent the higher-order statistics

of the "good" points, but also to have a learning algorithm which allows the parameters controlling the search distribution to be adjusted so that this distribution comes to capture the regularities of the set of "good" points. We now describe an algorithm which shows some promise in this regard.

5.2 REINFORCE/MENT Algorithms

We have performed extensive simulations of REINFORCE algorithms in networks having nontrivial connectivity, and these experiments have demonstrated that such algorithms are not successful in capturing the regularities of a set of output patterns leading to high reinforcement, even when the network is potentially capable of representing these regularities. One indication of their inability to capture such regularities is that they always converge to a single choice of output even when several output patterns all lead to the same maximum reinforcement value, as we demonstrate below. The end result is that having nontrivial connectivity in the network does not diminish susceptibility to convergence to false optima when REINFORCE is used. This has led us to devise a novel variant which we call the *REINFORCE/MENT* algorithm. It combines the use of the REINFORCE approach with entropy maximization.[2] The use of entropy maximization is designed to help keep the search alive by preventing convergence to a single choice of output, especially when several choices all lead to roughly the same reinforcement value.

We begin with some observations which apply in general to any situation where REINFORCE algorithms may be derived, and then we specialize to the case appropriate for our application, a network of Bernoulli logistic units having no (nonconstant) external input. Let n_O denote the number of output units in the network. Given an n_O-tuple $\boldsymbol{\xi}$, let

$$h(\boldsymbol{\xi}, \mathbf{W}, \mathbf{x}^{\mathcal{N}}) = -\ln Pr\left\{\mathbf{y}^{\mathcal{N}} = \boldsymbol{\xi} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\}. \tag{20}$$

Then

$$E\left\{h(\mathbf{y}^{\mathcal{N}}, \mathbf{W}, \mathbf{x}^{\mathcal{N}}) | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} = -\sum_{\boldsymbol{\xi}} Pr\left\{\mathbf{y}^{\mathcal{N}} = \boldsymbol{\xi} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} \ln Pr\left\{\mathbf{y}^{\mathcal{N}} = \boldsymbol{\xi} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\}, \tag{21}$$

which is the entropy of the output $\mathbf{y}^{\mathcal{N}}$ of the network, given the particular input pattern $\mathbf{x}^{\mathcal{N}}$ and weights $\mathbf{W}$. Thus, if we run the net with input $\mathbf{x}^{\mathcal{N}}$ and weights $\mathbf{W}$ and obtain output $\mathbf{y}^{\mathcal{N}}$, the quantity $h(\mathbf{y}^{\mathcal{N}}, \mathbf{W}, \mathbf{x}^{\mathcal{N}})$ is an unbiased estimate of this entropy.

Now we establish a key result. For the i^{th} unit in the network we let $\mathbf{w}^i$ denote the vector of weights on its incoming lines and we let $\mathbf{x}^i$ denote the pattern of input to that particular unit. For the Bernoulli logistic units we use here, s_i is the inner product of $\mathbf{w}^i$ and $\mathbf{x}^i$, but this notation applies more generally to any form of internal computation within the unit. Its random output value y_i is drawn from the discrete distribution having probability mass function g_i, where

$$g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = Pr\left\{y_i = \xi | \mathbf{w}^i, \mathbf{x}^i\right\} \tag{22}$$

[2] The suffix *MENT* stands for *M*aximization of *ENT*ropy.

For the special case of a Bernoulli logistic unit,

$$g_i(\xi, \mathbf{w}^i, \mathbf{x}^i) = \begin{cases} p_i & \text{if } \xi = 1 \\ 1 - p_i & \text{if } \xi = 0, \end{cases} \tag{23}$$

where p_i is the result of passing the inner product of $\mathbf{w}^i$ and $\mathbf{x}^i$ through the logistic function, as given by equations (1) and (2).

Lemma 1 *Let $\boldsymbol{\xi}$ be an n_U-tuple whose i^{th} coordinate is ξ_i, for all $i \in U$. Then, for any feedforward network of stochastic units,*

$$Pr\left\{\mathbf{y} = \boldsymbol{\xi} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} = \prod_{i \in U} g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i). \tag{24}$$

Proof. We prove this by induction. Assume that the vector $\mathbf{x}$ is indexed by integers in such a way that all the unit outputs correspond to indices in the range $[a, b]$ and in such a way that that w_{ij} is nonzero only if $j < i$. Because the network is feedforward there necessarily exists such an indexing. With this indexing, the value of each y_i depends only on the values x_j for $j < i$.

Now for any $k \in [a, b]$, let $\mathbf{y}^{(k)}$ denote the vector $(y_a, y_{a+1}, \ldots, y_k)$ and let $\boldsymbol{\xi}^{(k)}$ denote the vector $(\xi_a, \xi_{a+1}, \ldots, \xi_k)$. Clearly,

$$\begin{aligned} Pr\left\{\mathbf{y}^{(a)} = \boldsymbol{\xi}^{(a)} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} &= Pr\left\{y_a = \xi_a | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} \\ &= g_a(\xi_a, \mathbf{w}^i, \mathbf{x}^i), \end{aligned}$$

and to establish the induction step we assume that

$$Pr\left\{\mathbf{y}^{(k)} = \boldsymbol{\xi}^{(k)} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} = \prod_{i=a}^{k} g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i) \tag{25}$$

for some $k \in [a, b)$. Then

$$\begin{aligned} Pr\left\{\mathbf{y}^{(k+1)} = \boldsymbol{\xi}^{(k+1)} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} &= Pr\left\{y_{k+1} = \xi_{k+1} | \mathbf{y}^{(k)} = \boldsymbol{\xi}^{(k)}, \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} Pr\left\{\mathbf{y}^{(k)} = \boldsymbol{\xi}^{(k)} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} \\ &= g_{k+1}(\xi_{k+1}, \mathbf{w}^{k+1}, \mathbf{x}^{k+1}) \prod_{i=a}^{k} g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i) \\ &= \prod_{i=a}^{k+1} g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i), \end{aligned}$$

and the induction step is established. Therefore the Lemma is proved.

Note that this result is obvious for a team, in which all the output values are selected independently, but the Lemma shows that it holds more generally. It follows immediately from this Lemma that

$$-\ln Pr\left\{\mathbf{y} = \boldsymbol{\xi} | \mathbf{W}, \mathbf{x}^{\mathcal{N}}\right\} = -\sum_{i \in U} \ln g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i). \tag{26}$$

Thus, for a network having no hidden units, so that $\mathbf{y}^{\mathcal{N}} = \mathbf{y}$, the quantity

$$h(\mathbf{y}^{\mathcal{N}}, \mathbf{W}, \mathbf{x}^{\mathcal{N}}) = -\sum_{i \in U} \ln g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i) \tag{27}$$

is an easily computed unbiased estimate of the entropy of the output of the network as a function of its input. For example, when the network consists of Bernoulli logistic units, equation (23) shows that all that is needed to compute h is for each unit to send to a central summation location either $\ln p_i$, if its output y_i is 1, or $\ln(1 - p_i)$, if its output y_i is 0.

Even if there are hidden units, we could define $h(\mathbf{y}, \mathbf{W}, \mathbf{x}^{\mathcal{N}})$ similarly, and it would be an unbiased estimate of the entropy of the *state* of the network given the input, but this turns out not to be as useful. When there are stochastic hidden units, which could serve as command cells, for example, such an unbiased estimate of the output entropy is not easy to obtain, since $h(\mathbf{y}^{\mathcal{N}}, \mathbf{W}, \mathbf{x}^{\mathcal{N}})$ involves a summation over all possible states of such units and cannot be computed readily from quantities currently present in the network.

For our purposes here, we consider the nonassociative case, for which $\mathbf{x}^{\mathcal{N}}$ is constant. Thus we may suppress any reference to $\mathbf{x}^{\mathcal{N}}$ in the above and define the function

$$h(\boldsymbol{\xi}, \mathbf{W}) = -\ln Pr\left\{\mathbf{y}^{\mathcal{N}} = \boldsymbol{\xi} | \mathbf{W}\right\}. \tag{28}$$

For a network having no hidden stochastic units, then,

$$h(\boldsymbol{\xi}, \mathbf{W}) = -\sum_{i \in U} \ln g_i(\xi_i, \mathbf{w}^i, \mathbf{x}^i) \tag{29}$$

serves as an unbiased estimate of the output entropy on a particular trial whenever the actual output obtained on that trial is $\mathbf{y}^{\mathcal{N}} = \mathbf{y} = \boldsymbol{\xi}$.

We thus restrict attention to feedforward networks in which every random unit is an output unit. We define a *simplex* network to be a maximally connected feedforward network having no nonconstant input and no hidden units. An example of a simplex network is given in Figure 4. For such a network there is an ordering of the units such that every unit has a connection to all higher-numbered units but to no lower-numbered units.[3] In addition, each unit has a bias input.

Insert Figure 4 about here.

---

[3]Our use of the term *simplex* to describe such a network is based on the fact that the topological simplex can be defined combinatorially in a related fashion, based on such an ordering property. To see the relationship, note, for example, that a 3-node simplex network can be laid out in the form of a triangle and a 4-node simplex network can be laid out in the form of a tetrahedron.

A REINFORCE/MENT algorithm for optimizing the function J defined on the output patterns $\mathbf{y}^{\mathcal{N}} = \mathbf{y}$ of a network having no stochastic hidden units is obtained by using a REINFORCE algorithm in which the reinforcement signal is given by

$$r = J(\mathbf{y}^{\mathcal{N}}) + \varepsilon h(\mathbf{y}^{\mathcal{N}}, \mathbf{W}), \tag{30}$$

where $\varepsilon > 0$ is a parameter of the algorithm and h is computed via equation (27). For Bernoulli logistic units, a reinforcement comparison version of this algorithm is the following. After each trial, each weight w_{ij} is incremented by

$$\Delta w_{ij} = \alpha(r - \overline{r})(y_i - p_i)x_j, \tag{31}$$

where $\alpha > 0$ is the learning rate, r is computed using equation (30), and $\overline{r}$ is computed using equation (15).

Thus a REINFORCE/MENT algorithm is just a REINFORCE algorithm in which the overall reinforcement signal combines the "external" reinforcement (here given by J) with an "internal" reinforcement which is intended to reward variety. This internal contribution to the reinforcement has expected value proportional to the entropy of the distribution of output vectors produced by the reinforcement-learning network. This has the effect that the network is willing to sacrifice some performance to achieve the higher entropy enjoyed by continuing to explore. More importantly, as the experiments below demonstrate, it has the effect that if, during its exploration, the network discovers that there are a number of output patterns which lead to high reinforcement, the network weights will attempt to capture the regularities of all these points. This then biases future searching toward new points which share these regularities. In cases where it is appropriate to search in a hierarchical or modular fashion, the network can discover and exploit this organization of the search space, assuming that the particular network architecture used is capable of representing it.

To gain a better understanding of the behavior of the algorithm, consider the case when J is constant and assume that reinforcement comparison is used. Then, whenever a relatively unlikely point is sampled, the weights are adjusted to make this point more likely in the future; whenever a point of high likelihood is sampled, the weights are adjusted to make this point less likely in the future. The overall effect when J is not constant is that the algorithm trades off some performance (measured by J) in order to make the entropy higher.

The actual algorithm we used in the experiments whose results are reported in Table 1 was a slight modification of that given by equation (31), based on the use of units whose output values are -1 and +1 rather than 0 and 1. For uniformity of presentation, however, we will discuss the results as though 0/1 units were used. The weight update algorithm for this version is given by

$$\Delta w_{ij} = \alpha(r - \overline{r})(y_i^* - p_i)x_j, \tag{32}$$

where

$$y_i^* = \frac{y_i + 1}{2} \tag{33}$$

and all other quantities are the same as in equation (31). We found that this led to faster learning than when 0/1 units were used. In order to check that this speedup was not due simply to a fortuitous match between the -1/+1 representation used and the particular problems studied, we also ran some of the experiments using 0/1 units and two alternative algorithms. One algorithm we tried updates all weights using

$$\Delta w_{ij} = \alpha(r - \overline{r})(y_i - p_i)(2x_j - 1). \tag{34}$$

This algorithm seeks weights appropriate for 0/1 units but makes the weight changes as if input to any unit comes from -1/+1 units. Another algorithm we tried updates each bias according to equation (31), and updates all other weights using

$$\Delta w_{ij} = \alpha(r - \overline{r})(y_i - p_i)(x_j - \overline{x}_j), \tag{35}$$

where $\overline{x}_j$ is an exponentially weighted trace of past values of x_j. This conforms to an approach recommended by Sutton (1988b) to accelerate learning by decorrelating the learning of the biases from the learning of the other weights. We found that all 3 algorithms, pure REINFORCE/MENT using -1/+1 units and the two modifications just described for 0/1 units, performed essentially the same, and all were much faster than pure REINFORCE/MENT applied to 0/1 units.

5.3 Preliminary Demonstrations

To explore the effect of adding the entropy term, we ran both REINFORCE and REINFORCE/MENT on two problems involving a 2-unit simplex architecture. In the first experiment the function to be optimized is defined by

$$J(0,0) = J(1,1) = 10 \quad \text{and} \quad J(0,1) = J(1,0) = 0, \tag{36}$$

as illustrated in Figure 5. Treating J as the network's "reward" (i.e., disregarding the added entropy term), this corresponds to rewarding the network only when both units produce the same output. When REINFORCE is used, the network will always "go deterministic," converging to a single optimal choice of output, either $(0,0)$ or $(1,1)$, on any particular training run. On the other hand, using REINFORCE/MENT with $\varepsilon = 1$ and $\alpha = 0.05$ leads to weights like those shown in Figure 5 after several hundred trials. These weights have the effect that the two high-payoff points are each generated with probability very close to 1/2 while the probability of generating the remaining two points is very close to zero.

Insert Figure 5 about here.

In the second experiment the function to be optimized is defined by

$$J(0,1) = J(1,0) = J(1,1) = 10 \quad \text{and} \quad J(0,0) = 0, \tag{37}$$

as illustrated in Figure 6. Thus the network receives optimal reward for any output pattern having at least one 1. Once again, REINFORCE always causes the network to converge to one of the three optimal choices of output on any particular training run. On the other hand, using REINFORCE/MENT with $\varepsilon = 1$ and $\alpha = 0.05$ leads to weights like those shown in Figure 6 after several hundred trials. These weights have the effect that the three high-payoff points are each generated with probability very close to 1/3 while the probability of generating $(0, 0)$ is very close to zero. The way this works is as follows: The bias on the first unit causes it to generate a 0 about 1/3 of the time and a 1 about 2/3 of the time; the remaining weight and bias cause the second unit to almost always generate a 1 when the first unit produces a 0 and to generate a 0 or 1 with roughly equal probability when the first unit produces a 1.

Insert Figure 6 about here.

To appreciate the significance of these results, consider a high-dimensional optimization problem in which, early in the search, there is a pair of bit positions for which the payoff, when viewed as a noisy function of only those bit positions, is like, say, the first problem. That is, early in the search, approximately equally high payoff is received if they match, but much lower payoff is received if they don't. An algorithm like REINFORCE will then essentially rule out either $(0, 0)$ or $(1, 1)$ for these bits even though there is no good reason to do so. In fact, there may be good reason not to do so, because it may happen that one or the other may turn out to be much better when used in conjunction with other choices discovered much later in the search. REINFORCE/MENT, on the other hand, tries to keep all the options open during the search.

There is another point that is nicely illustrated by the weights obtained in the second experiment. Note that for a team, the use of bias decay amounts to an alternative way to incorporate a force toward increased entropy into the learning algorithm. The essential difference between the use of bias decay and REINFORCE/MENT applied to a team is that the weight changes prescribed by REINFORCE/MENT on any particular trial depend on the actual behavior of the network on that trial, while weight decay does not. On the average, however, the effect is the same. One might thus imagine that weight decay is a reasonable alternative to the REINFORCE/MENT algorithm in any interconnected network, not just a team network. However, it is clear that if the bias of the first unit in the network of Figure 6 were to decay toward zero, that unit would produce a 0 or 1 with roughly equal probability, which would not allow the three high-payoff points to be generated with equal probability.

5.4 Results on the Optimization Problems

Insert Table 1 about here.

A summary of the results of the experiments we performed using REINFORCE/MENT with a simplex architecture on the six optimization problems is included in Table 1. The parameter ε was fixed at 1 for all the problems, but some variation in the learning rate α was explored. As with each of the team algorithms, the table shows for each problem the particular value of learning rate used for all the runs using the algorithm on that problem and the rounded mean run length (number of function evaluations) to reach a global optimum for that set of runs. In each case this mean is based on 50 runs of each algorithm/problem combination.

5.4.1 The Abstract Problems

It is interesting to note that REINFORCE/MENT did not perform as well as the team algorithms on the four abstract problems, but it easily outperformed the best of them on the hierarchical graph partitioning problems. Like the team algorithms, it performed essentially as well on Porcupine as on One-Max. However, unlike the team algorithms, it performed better on Two-Max than on One-Max. We believe that the reason for this last result may be related to its strengths on problems like the hierarchical graph partitioning problems. The fact that there are two local optima in the Two-Max problem may give the algorithm a better chance of finding the global maximum because some of what it learns exploring around the false peak generalizes more readily to the global peak; in particular, it may be learning that making all the bits identical is good.

These observations should be regarded as purely speculative; we have not yet analyzed in any detail the performance of simplex network REINFORCE/MENT on these abstract problems because we have chosen to concentrate on its performance in the hierarchical graph partitioning problems, which provided the main motivation for devising it in the first place. One likely reason why it may be inferior to the team algorithms on simpler problems is that a simplex network having n_O units has $n_O(n_O + 1)/2$ weights while a team network having the same number of units has n_O weights. While the team tries to fit a 20-parameter distribution to its current view of the high-payoff parts of the space for these abstract problems, the corresponding simplex network tries to fit a 210-parameter distribution to the same data. The network having more weights must, in a sense, rule out many more hypotheses about what it is that the high-payoff points seen so far have in common.

5.4.2 The Hierarchical Graph Partitioning Problems

As shown in Table 1, REINFORCE/MENT in a simplex architecture was able to find one of the two global maxima for the MLC-32 problem in an average of just over 1000 function evaluations, and it was able to find one of the two global maxima for the MLC-64 problem in an average of just over 4900 function evaluations. To gain further insight into the performance of REINFORCE/MENT on these multilevel hypercube graph problems, we also performed some more detailed analysis of its behavior over the course of seeking a solution.

In what follows, we illustrate those aspects of the behavior we analyzed for REINFORCE/MENT in a simplex network by contrasting it with the corresponding behavior of REINFORCE with bias decay in a team network, the one other algorithm/architecture combination which has also successfully solved these problems in our experiments. Our intent here is simply to demonstrate that the former combination does appear to conduct its search in an manner appropriate for this type of problem, in contrast to other techniques which might also be able to eventually find a solution as well.[4]

One way to determine if an algorithm for exploring a space is truly finding regularities shared by many points giving relatively high payoff is to see what it generates after finding a global maximum (or, in a broader context, after finding a "very good" point, in some sense). In particular, since there are 2 equally good solutions for each of these problems, we did not terminate a run when one was found; instead, we let the network continue to generate trial points to see how long it would take until the other solution was found as well. The result for MLC-32 was that, over the 50 runs, an average of only 145 additional function evaluations was required before the second optimal point was generated. Thus, on the average, *both* solutions were found within 1161 function evaluations. For MLC-64 it took an average of 260 additional function evaluations to find the second optimal solution, so that both solutions were found within 5161 function evaluations on the average. Although we did not try this with the team algorithms incorporating weight decay, there is no doubt that it would take them at least as long as, and almost certainly much longer than, it takes them to find the first optimal solution. The team obviously cannot represent the necessary regularities for these problems.

Figure 7 shows a plot of the value of J as a function of trial number for one typical run of REINFORCE/MENT on the MLC-32 problem. Note that the value of J at a global optimum is 0 for the MLC-32 graph since it can be partitioned with no edge cuts. Thus the plot shows that a global optimum was first found near trial number 1000. It also shows that some of the subsequent trials led to generation of global optima as well, with such points being generated with ever-increasing frequency. Furthermore, although this plot does not show which points are generated on each trial, it turns out that both optimal points were generated numerous times during this run.

Insert Figure 7 about here.

For comparison, Figure 8 shows a corresponding plot of the behavior of one run of team REINFORCE with bias decay on the same problem. This particular run is atypical in the sense that a global optimum was found much sooner than in the average

[4]Note that the architectures and algorithms used in these comparisons can be combined in two other ways. While these other combinations may be capable of solving these graph-partitioning problems as well, we did not experiment with them because it is clear that they do not have the properties we describe here. The representational limitation of the team architecture is obviously not affected by the algorithm used, and, as we have noted earlier, the use of bias decay or weight decay together with REINFORCE does not necessarily help identify the appropriate regularities.

case (around trial number 1000 rather than trial number 5571), but we use it here to illustrate the behavior over a comparable time interval. Unlike the results using REINFORCE/MENT with a simplex network, this algorithm/architecture combination does not lead to ever-more-frequent generation of high-payoff points. Furthermore, only one of the two optimum points is generated.

Insert Figure 8 about here.

In order to examine how well the simplex network using REINFORCE/MENT learns to represent the regularities of high-payoff points as it explores, we also studied both the distribution of points it generated and the weight matrix it evolved throughout single runs on the MLC-32 problem. Because the essence of solving this problem is discovering the eight highly connected 4-node clumps in the graph, we concentrated on examining how well it seemed to represent and exploit this information. Examination of the weights evolved shows that, well before the network first generates an optimal point, the within-clump weights begin to take on positive values while the between-clump weights stay closer to zero, in general. This allows exploration to proceed in such a way that whole clumps can be freely placed on either side of the partition, which represents an appropriate strategy for dealing with such graphs, effectively reducing the size of the search space from 32 dimensions to 8 dimensions for MLC-32. We believe that the success of REINFORCE/MENT on such problems rests on what can be roughly characterized as its ability to find and experiment with placement of these clumps; a corresponding strategy is also behind the well-known graph partitioning algorithm of Kernighan and Lin (1970).

Detailed understanding of the search behavior from the weights alone is not that easy, however. This is because all the weights tend to have relatively small magnitudes in the 32-unit simplex network used for these experiments since, in any n-unit simplex network, the n^{th} unit receives input from $n+1$ sources. Thus we also examined the distribution of output values selected by the network within each clump during individual runs. In particular, we plotted histograms, one for each clump, showing the number of 1s generated for that clump during periods consisting of 100 consecutive function evaluations. Figure 9 shows a subset of the histograms generated for the same run of REINFORCE/MENT used for Figure 7. The histograms for the first 100 trials show the approximately bell shape appropriate for a sum of 4 Bernoulli random variables (in this case, with $p = 1/2$, since all weights are initialized at 0). The histograms for trials 401-500 have already begun to flatten out, indicating a preference for the all-0 or all-1 cases. The remaining histograms show that this tendency continues as more sampling is done, so that by the final 100 trials of this run, which was terminated after 1500 trials, the network has evolved a very strong preference for having all values within a clump be the same, but almost no preference for whether they all be 0 or all be 1. Thus, throughout the run, the search evolves in such a way that within-clump coordination

becomes more and more likely. In short, the network discovers the significance of the clumps and allows exploration to proceed using clump-level manipulations.

Insert Figure 9 about here.

For comparison, Figure 10 shows the corresponding histograms for the same run of team REINFORCE with bias decay depicted in Figure 8. All the histograms have the approximately bell shape appropriate for a sum of 4 Bernoulli random variables, regardless of how far the run has proceeded, since the team network cannot coordinate the behaviors of individual units. The histograms for the final 100 function evaluations show that the network has a fairly strong preference for having all units corresponding to nodes in the first 4 clumps on one side output a 1 while having all other units output a 0. Thus this network has not captured the regularities characteristic of a wide range of high-payoff points.

Insert Figure 10 about here.

Note that we have omitted any mention of the precise mapping of nodes in the graph to units in the simplex network in this discussion. While there is an obvious symmetry among all members in a team network, there is a definite ordering to the computation performed within a simplex network. This leads to the possibility that assigning different problem representation roles to the individual units in such a network might lead to different results when performing the search for an optimum. We investigated this for the graph partitioning problems by running a number of trials in which the mapping between graph nodes and network units was mediated by randomly generated permutations. What we found was that there was virtually no difference in the results.

Finally, another interesting result we observed is that not only did REINFORCE/MENT in the simplex network find solutions to the graph partitioning problems more rapidly on average than any other algorithm we tried, it was also much more consistent in its ability to find solutions relatively quickly. For this algorithm, the standard deviation of the length of time to find a global maximum over the 50 runs was 415 for MLC-32 and 1368 for MLC-64. In contrast, the standard deviations for the team using REINFORCE with weight decay were 3580 for MLC-32 and 4379 for MLC-64; for the version using $y_i - \overline{y}_i$ eligibility, the standard deviations were 4648 for MLC-32 and 5013 for MLC-64. The longest time for any run of the simplex network using REINFORCE/MENT was 1787 on MLC-32 and 9707 on MLC-64, while the longest times for the two successful team algorithms were in the range 17,000-26,000 for these two tasks.

6 Discussion

In this paper we have described the results of applying a number of variants of REINFORCE algorithms to some optimization problems studied by Ackley. These algorithms include team algorithms as well as a novel network algorithm we have derived which incorporates an entropy maximization strategy. The relatively good performance of the simple single-bit correlation team algorithms on many of these problems was surprising to us and may be an indication that these particular problems are not as challenging as one might have hoped.

However, we should point out that there is another function, studied by Ackley, called the *Trap* function, for which none of the algorithms we have discussed here is successful at finding the global maximum. This function is essentially a version of Two-Max in which a much larger proportion of the space leads uphill to the false peak. In Ackley's studies, the only algorithm which was at all successful at solving this type of problem was a simple hillclimber which is restarted after reaching any peak.[5] The useful exploration performed by such an *iterated hillclimber* is really only in the restarts; if it begins in the right region, it will succeed, but not otherwise. All the other algorithms he studied, and all the algorithms investigated here, involve some amount of more global sampling which prevents them from taking the necessarily myopic point of view necessary to succeed at such tasks. It is interesting to note that a still more difficult function can be created by adding "porcupine quills" to the entire space, or just to the vicinity of the global peak, leading to a function that an iterated hillclimber would find essentially impossible to optimize as well.

Another aspect of the relation of our work to Ackley's which we should point out is that he was careful to use uniform parameter settings for each algorithm throughout his experiments. This is obviously the only fair way to optimize a black-box function. In contrast, not all the parameters of the algorithms whose results we have reported have been fixed across problems, although most have. In particular, only one parameter, the learning rate, was varied across problems, and in many cases the results we have reported here involve relatively little variation of this parameter across problems for any fixed algorithm. To be able to claim that any of the algorithms here are truly useful black-box optimization algorithms we would need to conduct more studies using one fixed value of the learning rate; alternatively, one might imagine adding a meta-level adaptation designed to determine useful settings of these usually experimenter-controlled parameter settings.

An interesting potential application of Lemma 1 that we did not take advantage of in these studies is to avoid extensive Monte Carlo simulations by directly computing the probability that a network having all random units as output units would produce a particular output. This can be used for a pattern already generated, as when a network

[5] Ackley's SIGH algorithm could also find the global peak in the special case when the global and false peak were at bitwise complementary points, because its distribution of trial points was biased in favor of such symmetry; with more general placement of these two peaks, SIGH failed like all the other non-hillclimbers.

produces a global optimum for a function and it would be useful to know whether that happened as a fluke or was destined to happen soon anyway, and it can also be used to test the network on a pattern without having to wait for it to be generated. By simply "loading" the pattern in as if it were the actual output generated and then performing the computation of the p values, one can determine through the use of Lemma 1 the probability that the network would have generated this pattern on its own.

Of all the work described here, the part we are most interested in investigating further is that dealing with the REINFORCE/MENT algorithm. There are several aspects of REINFORCE/MENT which could be explored further. It would be helpful if it did not have to be limited to cases where all random units are output units, and it would also be interesting if a version could be devised which allows more symmetrical interactions among the units, perhaps involving the use of something more like an asynchronous update strategy or some type of settling behavior. Although we have not investigated it in any detail, it appears that the reinforcement learning algorithm for a Boltzmann machine described by Hinton (1989) is one such way to use more symmetrical interactions, although it would be interesting to find others which do not require a complete annealing process just to generate a single trial point.

More generally, it would be worthwhile to understand the relation between the REINFORCE/MENT approach and other approaches which also incorporate an entropy formulation, such as simulated annealing. Clearly, ε is essentially a temperature parameter. Initial investigation of this question has led us to suspect that while there are some important similarities which might be useful to exploit, there also appear to be some differences. Whether these differences are significant or merely superficial must await further analysis.

Finally, an interesting extension of the REINFORCE/MENT approach which we have begun to explore is its possible applicability in associative reinforcement learning, where the input presented to the network is not constant. In this case, adding to the external reinforcement the entropy of the entire state of the network as a function of the input, computed according to equation (27), seems intuitively reasonable. The idea is that this entropy is best optimized by finding a set of internal representations, viewed as patterns of activity over the hidden units, any of which could potentially do the job required for the particular input presented, without committing prematurely to any particular one. We have studied the use of this algorithm in the 2-hidden-unit XOR task (using all Bernoulli logistic units) and found that it succeeds. The external reinforcement used in this task is 1 if the output is correct and 0 otherwise. This form of reinforcement almost always leads to failure for the statistically gradient-following pure REINFORCE algorithm because of the presence of attractive local maxima, although more sophisticated reinforcement functions can be devised which allow pure REINFORCE to solve the XOR problem. Another algorithm which succeeds on such tasks, using the same success/failure reinforcement signal, is the associative reward-penalty (A_{R-P}) algorithm (Barto & Anderson, 1985). Our preliminary studies comparing the behavior of REINFORCE/MENT with A_{R-P} on this task show that REINFORCE/MENT may be somewhat slower, but this is probably because it does not incorporate the absolute

standard of reinforcement which A_{R-P} does.

7 References

Ackley, D. H. (1987). *Stochastic iterated genetic hillclimbing.* Ph.D. Dissertation, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. Also available as: *A Connectionist Machine For Genetic Hillclimbing.* Norwell, MA: Kluwer.

Barto, A. G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology, 4*, 229-256.

Barto, A. G. & Anandan, P. (1985). Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics, 15*, 360-374.

Barto, A. G. & Anderson, C. W. (1985). Structural learning in connectionist systems. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, 43-53.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics, 13*, 835-846.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: W. H. Freeman.

Goldberg, D. E. & Holland, J. H. (Eds.) (1988). Special Issue on Genetic Algorithms, *Machine Learning, 3*, nos. 2/3.

Hinton, G. E. (1989). Connectionist learning procedures. *Artificial Intelligence, 40*, 185-234.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press.

Hopfield, J. J. & Tank, D. W. (1985). Neural computation of decisions in optimization problems. *Biological Cybernetics, 52*, 141-152.

Kernighan, B. W. & Lin, S. (1970). An efficient heuristic technique for partitioning graphs. *Bell Systems Technical Journal, 49*, 291-307.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220*, 671-680.

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

Narendra, K. S. & Thathatchar, M. A. L. (1989). *Learning Automata: An Introduction.* Englewood Cliffs, NJ: Prentice Hall.

Sutton, R. S. (1984). Temporal credit assignment in reinforcement learning. Ph.D. Dissertation, University of Massachusetts, Amherst (also COINS Technical Report 84-02).

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

Sutton, R. S. (1988b). *NADALINE: A normalized adaptive linear element that learns efficiently* (Technical Report 88-509.4). GTE Laboratories Inc., Waltham, MA.

Sutton, R. S. & Pinette, B. (1985). The learning of world models by connectionist networks. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, 54-64.

Williams, R. J. (1986). Reinforcement learning in connectionist networks: a mathematical analysis (Technical Report 8605). University of California, San Diego, Institute for Cognitive Science.

Williams, R. J. (1987). A class of gradient-estimating algorithms for reinforcement learning in neural networks. *Proceedings of the First Annual International Conference on Neural Networks, II*, pp. 601-608.

Williams, R. J. (1988). Toward a theory of reinforcement-learning connectionist systems (Technical Report NU-CCS-88-3). Northeastern University, Boston, MA.

Williams, R. J. & Peng, J. (1989). Reinforcement learning algorithms as function optimizers. *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, Vol. II, 89-95.

Architecture	Algorithm		
	Reinforcement Factor	Eligibility	Weight Decay
Team	$r - \overline{r}$	$y_i - p_i$	No
Team	$r - \overline{r} - \beta$	$y_i - p_i$	No
Team	$r - \overline{r}$	$y_i - p_i$	Yes
Team	$r - \overline{r}$	$y_i - \overline{y}_i$	No
Team	$r - \overline{r} - \beta$	$y_i - \overline{y}_i$	No
Team	$r - \overline{r}$	$y_i - \overline{y}_i$	Yes
Simplex	REINFORCE/MENT		
Best found by Ackley for task			

Optimization Task											
One-Max		Two-Max		Porcupine		Plateaus		MLC-32		MLC-64	
α	Time	α	Time	α	Time	α	Time	α	Time	α	Time
0.009	130	0.004	149	0.009	139	0.002	465		∞		∞
0.01	123	0.004	147	0.01	120	0.002	446		∞		∞
0.05	112	0.005	273	0.05	119	0.01	335	0.4	5571	0.2	6643
0.009	9	0.004	102	0.009	9	0.002	435		∞		∞
0.009	10	0.011	38	0.009	10	0.004	314		∞		∞
0.05	6	0.3	16	0.1	6	0.01	234	0.4	5127	0.2	7356
0.002	335	0.001	203	0.001	364	0.001	862	0.007	1016	0.0015	4901
	19		35		357		494		2574		24905

Table 1: Summary of simulation results on the optimization problems.

Figure 1: The hierarchically structured graph MLC-32.

Figure 2: The hierarchically structured graph MLC-64.

Figure 3: A 5-unit team network. Every unit is an output unit, and the only adjustable weights are the biases, which can be treated as weights on the connections indicated by the arrows. The "presynaptic" signal on each of these connections is the constant 1.

Figure 4: A 4-unit simplex network. Every unit is an output unit. Among the adjustable weights are the biases, which can be treated as weights on the connections indicated by the arrows. The "presynaptic" signal on each of these connections is the constant 1.

Figure 5: A 2-unit simplex network and the optimization problem it has learned to solve using the REINFORCE/MENT algorithm. Ideally, the bias of the first unit should be 0 and the bias of the second unit should have half the magnitude of the weight between the units.

Figure 6: A 2-unit simplex network and the optimization problem it has learned to solve using the REINFORCE/MENT algorithm. Ideally, the bias of the first unit should be $\ln 2$ and the remaining two weights should have equal magnitude.

Figure 7: Plot of J as a function of trial number for a typical run of REINFORCE/MENT in a simplex architecture facing the MLC-32 graph partitioning problem.

Figure 8: Plot of J as a function of trial number for one run of REINFORCE with bias decay in a team architecture facing the MLC-32 graph partitioning problem.

Figure 9: Histograms showing the number of 1s generated in each of the 8 MLC-32 clumps during specific 100-trial periods for a typical run of REINFORCE/MENT in a simplex architecture. Clumps are ordered so that the first 4 should all be placed together in an optimal partition.

Figure 10: Histograms showing the number of 1s generated in each of the 8 MLC-32 clumps during specific 100-trial periods for one run of REINFORCE plus weight decay in a team architecture. Clumps are ordered so that the first 4 should all be placed together in an optimal partition. The solution found in this experiment clearly assigns each node in the first 4 clumps to the 1 side of the partition and the remaining nodes to the 0 side.