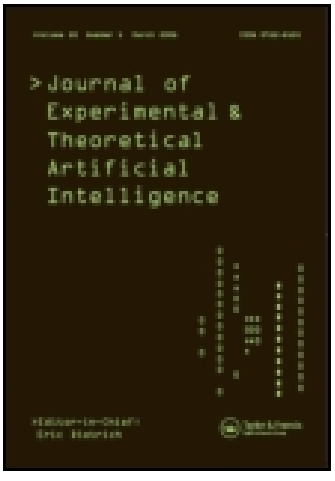

Journal of Experimental & Theoretical Artificial Intelligence
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/teta20

Learning nested concept classes with limited storage

DAVID HEATH

Published online: 10 Nov 2010.

To cite this article: DAVID HEATH (1996) Learning nested concept classes with limited storage, Journal of Experimental & Theoretical Artificial Intelligence, 8:2, 129-147, DOI: 10.1080/095281396147429

To link to this article: http://dx.doi.org/10.1080/095281396147429

J. Expt. Theor. Artif. Intell. 8(1996)129–147.

Learning nested concept classes with limited storage

DAVID HEATH, SIMON KASIF, RAO KOSARAJU,
STEVEN SALZBERG and GREGORY SULLIVAN

Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, USA

Abstract. Resource limitations have played an important role in the development and verification of theories about intelligent behaviour. This paper is a step towards answering the question of what effect limited memory has on the ability of intelligent machines to learn from data. Our analysis is applicable to many existing learning methods, especially those that incrementally construct a generalization by making repeated passes through a set of training data (e.g. some implementations of perceptrons, neural nets, and decision trees). Most of these methods do not store the entire training set, since they allow themselves only limited storage, a restriction that forces them to produce a compressed representation. The question we address is, how much (if any) additional processing time is required for methods with limited storage? We measure processing time for learning algorithms by the number of passes through a data set necessary to obtain a correct generalization. Researchers have observed that for some learning methods (e.g. neural nets) the number of passes through a data set gets smaller as the size of the network is increased; however, no analytical study that explains this behaviour has been published. We examine limited storage algorithms for a particular concept class, nested hyperrectangles. We prove bounds that illustrate the fundamental trade-off between storage requirements and processing time required to learn an optimal structure. It turns out that our lower bounds apply to other algorithms and concept classes as well (e.g. decision trees). We discuss applications of our analysis to learning and to problems in human perception. We also briefly discuss parallel learning algorithms.

Keywords: incremental learning, learnability

Received 6 July 1992; accepted 30 August 1992

1. Introduction

Resource limitations have played an important role in the development and verification of theories about intelligent behaviour. Particularly striking examples of this notion are such seemingly unrelated ideas as bounded rationality, bounded order perceptions, Feldman's 100-step metaphor, and others. Each of these theories puts a limit on the amount of memory, or the amount of inference, allowed to a rational agent. This paper takes a step towards answering the question of whether limited memory has an effect on the ability of intelligent machines to acquire structure from data. While our intuition suggests that any limitation in computational resources, such as memory, time, or degree of connectivity in a network, may have an effect on the

0952–813X/96 $12·00

computational capabilities of a model, an analytical study is important to quantify such limitations. An example of one such study is the classical work by Minsky and Papert (1988) on perceptrons, where it was shown that bounded-order perceptrons cannot compute simple predicates such as parity or connectivity in an image. This rigorous study strengthened the belief that perceptrons were probably too simple to be an accurate model of perception.

In our learning model, an incremental or on-line learning algorithm is one that constructs a generalization after one pass through a set of training data, and modifies it in subsequent passes to make it smaller or more accurate. Many perceptron-based methods, neural network algorithms, and incremental decision tree techniques fit this paradigm. Most of these methods do not store the entire set of training data. When processing is completed, the only thing they retain is a generalized data structure such as a tree, a matrix of weights, or a set of geometric clusters.

The phenomenom that stimulated our original interest in the subject is that fact that some learning architectures (e.g. neural nets using the backpropagation algorithm) often tend to require fewer training passes to learn a function when the size of memory is increased. We wanted to understand whether this behaviour was a characteristic of a particular algorithm, or an inherent feature of any learning algorithm with a restricted memory capacity.

Limited storage is an important consideration for several reasons. First of all, some classes of learning algorithms *always* use fixed storage. Neural net learning algorithms, for example, have a fixed number of nodes and edges, and only change the weights on the edges. Decision trees can in principle grow without bound, but in practice researchers have devised many techniques for restricting their growth (Quinlan 1986, Utgoff 1989). Second, limited storage is a realistic constraint from the perspective of cognitive modelling—human learning behaviour clearly must adhere to some storage limitations. Finally, there is experimental evidence that restricting storage actually leads to better performance, especially if the input data is noisy (Aha and Kibler 1989, Quinlan 1986). It also has been observed that when a neural network is trained using the back-propagation algorithm, increasing the size of the network may improve the speed of learning and at the same time degrade its predictive accuracy due to overfitting.

The problem we address in this paper is whether reducing the memory capacity of a learning algorithm will have an effect on the speed of learning. More specifically, the question we address is how many passes through a data set are required to obtain a correct generalization if we are only allowed enough memory to store the generalization. We also ask how speed increases as we give an algorithm additional storage.

There are several reasons for using the number of passes, rather than processing time, in this analysis. First of all, we believe that for some learning problems the relationship between memory size and the number of passes required is a more basic relationship than that between memory size and processing time. For example, the lower bound that we present below is independent of the learning algorithm, if we make just a few assumptions about the algorithm. This contrasts with processing time, which may vary widely between algorithms. Secondly, for some learning algorithms, such as certain implementations of neural networks, the number of passes through the data is a natural measure of learning complexity. That is, although theoretically we may have enough examples in the training set to learn a given concept correctly, we may need to present the training set to a given network many times before the network can accurately represent it.

Our results show that if a fixed-storage algorithm attempts to create a minimal-size concept structure, then it *cannot* generalize on-line without losing accuracy. We also show that by making a number of additional passes (depending on the number of concepts) through the data set, an algorithm can create an optimal structure (minimum in size and consistent with the training set). Our main goal is to demonstrate that there exists a *fundamental* trade-off between the storage available and the number of passes required for learning an optimal structure. There are few previous results comparable to ours on the number of passes required to learn a concept. Most previous analyses give estimates of the size of the input data set, not of the number of presentations required. This work is therefore an important step towards formalizing the capabilities of incremental versus non-incremental learning algorithms. Any algorithm that does not save all training examples is to some extent incremental, since upon presentation of new inputs the algorithm cannot re-compute a generalization using all previous examples. In addition to shedding light on learning from examples, our results may have implications for human perceptual tasks such as grouping of dot patterns (see Section 9.3).

The learning framework considered in this paper is that of computing generalizations in the form of geometric concept classes. Many important learning algorithms fall in this category, e.g. the standard perceptron learning algorithm (Rosenblatt 1959), some instance-based learning systems (Aha and Kibler 1989), most decision tree algorithms (Quinlan 1986) and hyperrectangle techniques (Salzberg 1991). We focus on the concept class defined by nested hyperrectangles, although some of our bounds are applicable to any algorithm that partitions a data set into convex regions, such as decision trees and perceptrons. A typical decision tree in a real-valued space contains at each non-leaf node a test of the form $x > k$, where x is an attribute and k is a constant. These tests are just axis-parallel hyperplanes that partition the examples into hyperrectangular regions. Hyperrectangles have been used in learning systems by several research groups for a variety of applications (see Michalski 1983, Salzberg 1989, 1990, Carpenter *et al.* 1991, Christiansen 1992, and others).

The learning model we consider has a fixed number of storage locations; i.e. it is not permitted to store and process all of the training examples at once. The limited storage requirement is applicable only during the learning (training) phase. In other words, our algorithms must operate incrementally, storing a limited number of intermediate results during each pass through a data set, and modifying the partially constructed

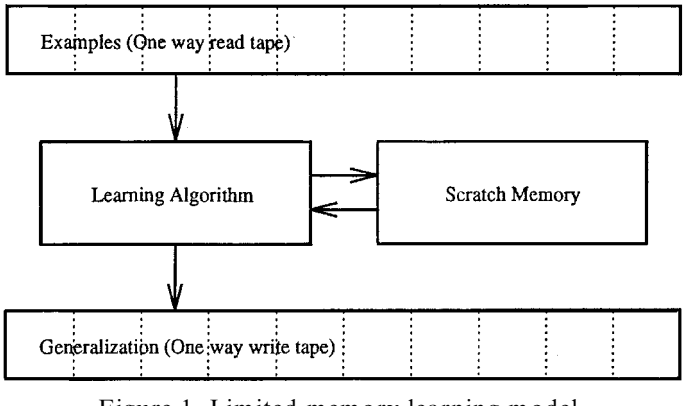

Figure 1. Limited memory learning model.

generalization in subsequent passes. For both cognitive and practical reasons, much experimental learning research has focused on the development of incremental algorithms (Utgoff 1989).

Our main results are stated informally below. The formal trade-offs are reported in Section 12.

- We report a general lower bound on the number of passes through a data set required by all algorithms learning concepts in the shape of convex partitions, including perception trees (Utgoff 1989), decision trees (Quinlan 1986), and hyperrectangles (Michalski 1983, Salzberg 1989, 1990, Carpenter *et al.* 1991, Christiansen 1992).
- We discuss several algorithms for learning nested concept structures with limited memory. Some of these methods are surprisingly complex, most notably the sampling algorithm of Section 9.3.
- We report a general technique that can be used to speed up algorithms that learn convex partitionings of space. This technique can be used by many learning algorithms. We present an analysis of the technique for a restricted case. We also discuss an application of our technique to a novel perceptual recognition problem.
- We analyse the parallel complexity of algorithms for obtaining generalizations in the shape of nested hyperrectangles.

2. Relevant work

Learning with limited memory has been studied recently in a theoretical framework by a number of researchers, including Haussler (1988) and Floyd (1989). This work generally considers the question of whether an algorithm can PAC-learn a concept using limited storage; i.e. whether it can with high probability learn a concept with high accuracy. One important difference between our work and the work by Haussler and Floyd is the fact that we require exact learning of the training set (no errors allowed). Our work follows the spirit of much experimental work, which often takes the approach of repeatedly presenting a training set until the algorithm is 100% consistent with the examples. Sometimes this is referred to as the 'loading problem'. Another significant difference (perhaps more significant) is the fact that we study trade-offs between (1) the size of the available memory and (2) the number of passes, while previous work has studied the trade-off between (1) memory size and (2) the probability of error. A study by Helmbold *et al.* (1989) discusses a different problem, namely learning with a memory efficient algorithm. There, the algorithm has access to all the examples, but the size of the generalization is restricted. In their model the examples are stored explicitly and assumed to be sorted. If the examples are presented in sorted order, then some of the problems we consider below become trivial. Finally, recent work by Frievalds *et al.* (1993) uses a recursion-theoretic approach to analyse learning with limited memory.

3. The limited memory model

The model of a limited memory algorithm we use in this paper is inspired by a model proposed by Munro and Paterson (1980). This model consists of four parts (see Figure 1). These are:

- The learning algorithm has some small amount of internal storage for keeping track of its state, but this storage is insufficient to store even a single input example.
- We can think of the examples as being written on a read-only one-way tape. The learning algorithm must examine the examples in the order in which they lie on the tape; moving backwards is not allowed. The one exception is that the learning algorithm can start over at the beginning of the tape as many times as it wants. However, each time it rewinds the tape, this begins another pass. Note that we do not assume that the examples are on the tape in any particular order. In fact, most of our results apply even if the order of examples is changed from pass to pass by an outside agent.
- The algorithm writes its generalization onto an output tape. It is not possible for the algorithm to read back the output tape, or to move the tape backwards.
- The limited memory that concerns us in this paper is a scratch memory store. It is used to store examples from the input tape that await further processing. This memory permits operations such as comparisons that involve more than one example.

It is important to note the distinction between the sizes of the various memories available to the learning algorithm (input, output, scratch, and internal). The model enforces no relationship between the number of examples (size of the input tape), the size of the generalization (size of the output tape), and the number of examples that can be stored in scratch memory. Without making the distinction between the different available memories, the reader may wonder how we can talk about cases in which the learning algorithm does not have enough memory to store all of the examples. In fact, all of the interesting cases occur when the size of the scratch memory is smaller than the number of examples. When the scratch memory is as large as the number of examples, we can simply read all of the examples into the scratch memory and do any processing in a single pass.

Why do we make such a distinction? The answer is that we are trying to formulate a model that encompasses some widely used learning algorithms. Specifically, we wish to model algorithms that are presented with training examples multiple times. For example, a particular implementation of a backpropagating neural network may be trained by presenting it with a set of data thousands of times. The actual computer that the network implementation is running on may have enough memory to store all of the examples, but that does not change the fact that the network itself uses a fixed amount of memory.

4. Nested hyperrectangles

Recent experimental research on learning from examples has shown that concepts in the shape of hyperrectangles are a useful generalization in a variety of real-world domains (Michalski 1983, Salzberg 1989, 1990, Carpenter *et al.* 1991, Christiansen 1992). In this work, rectangular-shaped generalizations are created from the training examples, and are then used for classification. Some algorithms, *e.g.* Salzberg (1989, 1990), Carpenter *et al.* (1991), Christiansen (1992), allow rectangles to be nested inside one another to arbitrary depth and classify new examples by the innermost rectangle containing them. Thus, inner rectangles may be thought of as *exceptions* to the surrounding rectangles. This learning model is called the Nested Generalized Exemplar

(NGE) model. Experimental results with this model have shown that it compares favourably with several other models, including decision trees, rule-based methods, statistical techniques, and neural nets (Salzberg 1989, 1990).

Independently of the experimental work cited above, Helmbold *et al.* (1989) have produced several theoretical results for nested rectangular concepts. In particular, they have developed a learning algorithm for binary classification problems that creates strictly nested rectangles, and that makes predictions about new examples on-line. They have proven that their algorithm is optimal with respect to several criteria, including the probability that the algorithm produces a hypothesis with small error and the expected total number of mistakes for classification of the first t examples. The algorithm applies to all intersection-closed classes, which include orthogonal rectangles in R^n, monomials, and other concepts. Their results assume that it is possible to classify the training examples perfectly using strictly nested rectangles.

Given that the learning community has found nested rectangles a useful concept class, and that the theoretical community has proven some further results about this same concept class, we have been led to investigate the ability of an algorithm to construct an optimal number of nested rectangles given only limited storage. As mentioned above, our results apply to other well-known concept classes, including those learned by decision tree algorithms and perceptrons.

5. Preliminaries

Here we define some of the basic terms that we will use in subsequent sections. An *example* is defined simply as a vector of real-valued numbers, plus a category label. For instance, we may be considering a problem where medical patients are represented by a set of real numbers including heart rate, blood pressure, and other attributes, and our task is to categorize the patients as 'inpatient' or 'outpatient'. For our purposes, an example is just a point in Euclidean space, where the dimensionality of the space is determined by the number of attributes measured for each example. We categorize points by using axis-parallel hyperrectangles, where each rectangle R_i is labelled with a category $C(R_i)$ such as 'outpatient'. A point is categorized by the innermost rectangle containing it. Figure 2 illustrates how categories are assigned. In the figure, lower-case letters indicate points belonging to categories a and b, and uppercase letter indicate the category labels of the rectangles. Notice that points not contained by any rectangle are assigned to category A, which corresponds to a *default* category. Only two rectangles are required to classify all the points in Figure 2.

The general problem definition is as follows: we are given n points in a d-dimensional space, and we are asked to construct a minimum (smallest possible) set of

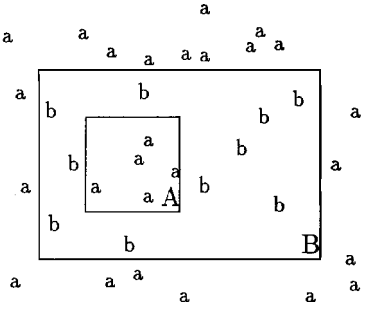

Figure 2. Categorizing points using rectangles.

strictly nested hyperrectangles that correctly classifies the set of examples. A precise definition is given in the next section. Clearly, in such a minimum representation no rectangle can be eliminated without causing misclassification (by definition), and additionally the number of rectangles is smaller or equal to any other minimal representation that uses nested hyperrectangles. We assume that there are just two classes.

Our algorithms learn by processing examples one at a time, in a random order. The algorithm is allowed to store no more than S of the examples, where S is chosen based on the number of examples and number of hyperrectangles needed for the minimal representation. In addition, the algorithm may have some additional constant amount of storage. On each *pass* through the data, the algorithm sees all of the examples exactly once. In most cases, the order of the examples on each pass is independent of the order on other passes.

Given these definitions, we would like to answer the following general question: how many passes p through the data are required to construct a minimum set of nested hyperrectangles? Below, we present several algorithms, and show how the number of passes required changes as a function of the amount of storage S and of the minimum number of rectangles R.

6. Definition of the learning problem

An instance of the nested rectangle problem is a set X of points in d-space, and a category identifier associated with each point. A solution to the problem is a set $\{R_1, \ldots, R_R\}$ of nested axis-parallel rectangles with the following properties:

1. Each rectangle is a closed region in d-space defined by $R_i = \{x \mid MIN_i \leqslant x \leqslant MAX_i\}$ where x, MIN_i, and MAX_i are d-vectors.
2. For $1 \leqslant i < R$, R_i contains R_{i+1}. That is, $MIN_i \leqslant MIN_{i+1}$ and $MAX_i \geqslant MAX_{i+1}$.
3. Each rectangle R_i has an associated category identifier $C(R_i)$.
4. The category of a point must be the same as the category of the smallest rectangle which contains the point, if the point is contained within any rectangles. We say a point x is contained in rectangle R_i if $MIN_i \leqslant x \leqslant MAX_i$. All points not contained in any rectangle must belong to the same category.

7. Conditions for learnability

There are some input sets which cannot be learned with nested axis-parallel rectangles. In this section, we discuss necessary and sufficient conditions for learnability. We say a set of examples is *learnable* if the nested rectangle problem for this set has a solution. Here we present a necessary and sufficient condition for the learnability of a set of examples.

Definition 1. *In a binary classification problem, an axis-parallel hyperrectangle with nonzero area is called a* blocking rectangle *if every edge of the hyperrectangle intersects points of both classes.* □

Theorem 7.1. *A set of points from two classes is learnable if and only if there does not exist a blocking rectangle for the set.*

We will not provide a formal proof of the theorem. In the following sections, we present algorithms for the nested rectangle problem that run to completion when there

is no blocking rectangle, establishing the *if* condition of the theorem. First we explain why a set of points is not learnable if it has a blocking rectangle. The proof follows from the following observations. If any set of points can be partitioned by a set of rectangles, then the rectangles also correctly partition any subset of the points. So, consider a set of points which define and lie on a blocking rectangle. By way of contradiction, assume there is set of rectangles which solve the nested rectangle problem for these points. Consider the largest rectangle in the set which excludes at least one point. Without loss of generality, assume this point belongs to class C_1. There must be a point in category C_2 which is also excluded by this rectangle. By the definition of the nested rectangle problem, these points must belong to the same category, which is a contradiction. Thus, any point set which contains a blocking rectangle cannot be learned with nested axis-parallel rectangles.

Theorem 7.1 identifies a necessary condition for learnability by nested rectangles. One implication of this theorem is that strictly nested axis-parallel rectangles cannot learn the logical EXCLUSIVE-OR function. However, one should note that we can easily obtain additional power simply by rotating the dataset by some random amount. With high probability, this trick will eliminate blocking rectangles and the data set will become learnable.

8. Static algorithm

First, we consider how the problem may be solved when enough memory is available to store every point. The algorithm is simple and is included for completeness. It also serves as a point of comparison against our truly memory-limited algorithms.

When all of the examples can be stored in memory ($S \geqslant n$), the nested rectangle problem can be solved by a modified plane sweep. We make one pass through the examples, during which we store all examples.

We sweep $2d$ axis-parallel hyperplanes, two for each axis. Each hyperplane defines two halfspaces: *inside* and *outside*. The intersection of the $2d$ *inside* halfspaces is a hyperrectangle. Initially we position the hyperplanes so that hyperrectangle R_0 is the smallest rectangle that contains all the examples. Let the category of R_0 (which is not a partitioning rectangle) be $C(R_0)$.

The computation proceeds in R steps. In each step, we find the next hyperrectangle, R_{i+1}, nested inside R_i, by sweeping each hyperplane inward (towards the *inside* halfspace it defines) until it intersects a point not belonging to category $C(R_i)$. In some steps, some hyperplanes may not move at all. The new positions of the hyperplanes define the hyperrectangle R_{i+1}, which is output.

Note that this algorithm requires that the points are sorted along all dimensions, which requires $O(dn \log n)$ time. Each pair of hyperplanes sweeps over n points, so the sweep takes $O(dn)$ time. Thus, the total time needed, including that for sorting, is $O(dn \log n)$.

Theorem 8.1. *Given n points in d-dimensional space, the nested rectangle problem can be solved with a single pass in* $O(dn \log n)$ *time and* $O(dn)$ *space.*

9. Limited memory algorithms

In this section we describe a number of algorithms for learning when the available memory is limited. Some of these methods are surprisingly complex, most notably the sampling algorithm of Section 9.3.

9.1. *Simple limited memory algorithm*

The static algorithm can easily be converted to work in a case where there is not enough memory to store all of the examples. First, suppose we have fixed memory sufficient to store two rectangles plus one additional example, but not all examples. We show that R passes will suffice to find R rectangles.

To find the outermost rectangle, our algorithm finds the maximum and the minimum point in each dimension. These $2d$ points define the edges of the first (outermost) hyperrectangle. Inductively, assume i rectangles have been constructed and the category of R_i is $C(R_i)$. Let $inside(R_i)$ be the set of points inside the ith rectangle. Our algorithm will use $2d$ storage locations to store the $2d$ points that define R_i while it is finding R_{i+1}. We now find the maximum and minimum point in each dimension for category $C(R_{i+1})$ that belong to $inside(R_i)$.

This can be done in one pass by storing, for each dimension, the smallest and largest point of category $C(R_{i+1})$ in $inside(R_i)$ seen so far. This will require an additional $2d$ points. We need one additional memory location to store each new point as it is presented to our algorithm. We revise our current estimate of the smallest and largest points (if necessary) and continue as each new point is processed.

Since we can find the next partitioning hyperrectangle in one pass, we can solve the partitioning problem with $4d+1$ storage locations in R passes.

Theorem 9.1. *Given* $4d+1$ *storage locations, it is possible to solve the d-dimensional nested rectangle problem in R passes, where R is the number of rectangles.*

This algorithm can be seen as a line adjustment algorithm similar to the standard perceptron algorithm, since we adjust our best estimate of R_i with each new point. Unlike the standard perceptron algorithm, it has the following characteristics.

- It is guaranteed to converge in R passes.
- Hyperplanes always move in the same direction.
- A hyperplane can make large adjustments towards its final location.
- The algorithm can classify input data that cannot be classified using the standard perceptron algorithm.
- A hyperplane defining rectangle R_{i+1} will not be adjusted until R_i has reached its final location.

Our main results, described below, illustrate that in many cases we can improve the performance of our algorithm and learn the concept structure correctly in far fewer passes. Additionally, if R is small with respect to n, then *at least* R passes are required to define the rectangles.

9.2. *Speeding up the limited memory algorithm*

In this section we show that with a simple modification of the static algorithm, we can design an algorithm for the nested rectangle problem that runs in fewer than R passes, where R is the number of rectangles. A more efficient and intricate scheme will be described in the next section.

As above, the outermost rectangle can be found in one pass with only $2d+1$ storage locations. Assume inductively that i rectangles have been found. We use $2d$ storage locations for maintaining a hyperrectangle W, which is a window outside of which we have found all rectangles $\{R_1, \ldots, R_i\}$, where R_i is innermost. (Note that W is always contained within R_i.) Initially, W contains all the examples (no rectangles have been found). All points outside W can be ignored while the algorithm positions one or more

rectangles within R_i. For each of the $2d$ hyperplanes that define W, we allot S_h memory locations, where S_h is defined to be $(S-(2d+2))/2d$, and S is the total amount of storage available. These locations will be used to find the S_h closest points (of any category) to the hyperplane that are inside W. All of these points can be found in one pass. Once they are in memory, the set of S_h points associated with each hyperplane is sorted by distance from the hyperplane.

We define an *alternation* to be a pair of points that belong to different categories and are adjacent in a sorted list. The alternations in each list can be found, and then matched up to find partitioning rectangles. If each list has at least one alternation, the outermost alternations in each list define a partitioning rectangle R_{i+1}. Every point stored that is not inside R_{i+1} is removed from the lists, since these points cannot define rectangles nested within R_{i+1}. Now, the outermost alternations in each list define R_{i+2}. We continue placing rectangles this way until at least one list has no alternations. Note that some list could have no alternations at the beginning of the pass. In this case, we simply find no rectangles in that pass. In any case, we redefine W as follows: for each list that still contains alternations, we move its associated hyperplane to the last alternation that was deleted (if any). We move every other hyperplane to the innermost point in its list. Because these lists have no alternations, they cannot generate partitioning rectangles. Since at least one list has no alternations, at least one hyperplane has moved inward by S_h points (*i.e.* at least S_h points have been excluded from W). Since no point that leaves W can re-enter, W will be empty in no more than

$$\left\lceil \frac{n}{S_h} = \frac{2dn}{S-(2d+2)} \right\rceil$$

passes.

Theorem 9.2. *Given S storage locations, it is possible to solve the d-dimensional nested rectangle problem in $O(nd/(S-2d-2)$ passes.*

In particular, when the number of rectangles R is larger than $O(\sqrt{nd})$, and $R = O(S)$, the number of passes is smaller than R. In other words, we can always solve the problem in $O(\sqrt{nd})$ passes irrespective of the number of rectangles.

9.3. *Limited memory with sampling*

All of the preceding algorithms work by finding alternations from the outside, working inwards. Below, we present an algorithm that uses a different strategy. This strategy allows us to find R rectangles in $O(\log n)$ passes when the amount of storage is slightly bigger than R (*i.e.* $S = R \log n$). This algorithm finds alternations (transitions from one category to another) on the real line, and is applicable to any convex partitioning of the line, such as that implicitly created by a decision tree. While learning in one dimension is a very special case for machine learning, the complexity of our algorithm demonstrates the difficulty of learning efficiently with limited storage even in one dimension.

In addition, the paradigm we describe here is applicable to a learning problem that has been studied by cognitive psychologists, e.g. Pylyshyn (1989) and Pashler and Badgio (1985). In this problem, the experimenter initially places N points along a line on a computer screen. Some points are red and some blue, and they form R alternating homogeneously coloured regions. These points are presented to the subject in random order, one point at a time, and the subject is requested to identify the transitions from blue to red regions (the alternations). At the end of each series of presentations (in which all N points are shown) the subject either identifies all the alternations or

requests another presentation. The subject is allowed to use a fixed number M of markers, where $M \geqslant R$. The limited number of markers and the limited memory capacity of the subject usually result in the subject requiring many presentations of the points before he can learn the structure of the regions. If the subjects use a version of the static algorithm presented in the previous section, they will track the outermost (that is, leftmost and rightmost) points of a specific colour. This is similar to the experiment reported by Pahler and Badgio (1985), where the subjects were told to name the highest digit in an array. However, we conjecture that if subjects are allowed to practise this task, they will develop a more sophisticated algorithm that will enable them to perform the task with fewer presentations of the data. Namely, we would suggest that subjects will use a heuristic version of the algorithm presented below. That is, the subjects will use the markers to partition the screen into regions, and then when the points are presented, they will attempt to find an alternation within each of the regions. Recall that an alternation is a pair of adjacent points that belong to different categories. Once they detect all the alternations, they can easily combine them to describe the overall concept structure.

Our algorithm for finding alternations works as follows. As a first step, we need to determine how many examples are in the training set, so we use the first pass to count the number of examples. Through the remaining passes, we maintain a set of intervals to which we restrict the search for alternations. We attempt to maintain a nearly constant number, I, of intervals. The derivation of the value of I will be given below.

Initially, the alternations can occur anywhere in the data set, so we create a single interval containing every example in the training set.

The general method we employ is to look for 1 or 2 alternations in each interval in each pass. We can use the simple limited memory algorithm from Section 9.1 on each interval to find the (at most) two outermost alternations in each interval, using $O(I)$ space. One of two things can happen to each interval. The interval may contain more than two alternations, in which case we shrink the interval in size just enough to exclude the outermost alternations, reducing the number of alternations in that interval by two. Or, the interval may have two or fewer alternations, in which case we empty the interval of alternations. When this happens, we discard that interval, as no further processing of it is necessary.

When we discard some intervals, we may end up with fewer than I intervals. When that happens, we use a limited storage splitting technique developed by Munro and Paterson (1980). It splits each interval into three subintervals, each of which contains no more than a constant fraction of the examples in the original interval. To split I intervals, it takes $O(I \log^2 n/S)$ passes, using S storage. (S will be determined below, but must obey $I \leqslant S \leqslant I \log^2 n$.)

The following schematic code fragment is executed repeatedly until all alternations have been found:

1. while the number of intervals is fewer than I, but more than 0
2. split each interval into 3 subintervals, using Munro and Paterson (1980)
3. check each interval for an alternation
4. discard intervals not containing alternations
5. End while
6. Detect one or two alternations from each interval

The method we use to split the intervals in step 2 is guaranteed to split each interval into a small constant number of subintervals, each of which contains no more than a

Table 1. Choosing the number of intervals I.

Space available	Passes needed	I
$S \leqslant \frac{R}{\log^3 n}$	$O\left(\frac{R}{S}\right)$	S
$\frac{R}{\log^3 n} \leqslant S \leqslant R \log n$	$O(\log^{3/2} n \sqrt{R/S})$	$\frac{\sqrt{RS}}{\log^{3/2} n}$
$R \log n = S$	$O(\log n)$	$\frac{R}{\log n}$

constant fraction of the interval's points. Since the intervals will be shrinking in size by a constant fraction each time step 2 is executed, the while loop will be iterated no more than $O(\log n)$ times.

The while loop guarantees that there are at least I intervals, and each one contains an alternation. Step (6) will find I alternations in one pass, and this step will be executed at most $2R/I$ times. $O(I)$ storage locations will be used to maintain the intervals.

Recall that in step 2, the splitting algorithm we use will require $O(I \log^2 n/S)$ passes, if it is given S storage. Steps 3 and 4 are done together in a single pass, without any additional storage. Since steps 2 through 4 are executed at most $O(\log n)$ times, a total of $O(I \log^3 n/S)$ passes will be needed for steps 2 through 4. Combining all of the steps, the algorithm uses a total of $O(2R/I + I \log^3(n)/S)$ passes with S memory locations. The number of passes is minimized when $I = \sqrt{2RS}/\log^{3/2} n$. We must maintain $1 \leqslant S/I \leqslant \log^2 n$, however. This will require that $S/\log^2 n \leqslant I \leqslant S$.

The choice of I depends on R and the memory/speed trade-off desired (see Table 1). In practice, we always have enough space for the rectangles ($S \geqslant R$) and therefore we can find all the alternations in $O(\log^{3/2} n)$ passes.

There are many situations in which the number of rectangles R is not known beforehand. In this case, a binary search technique can be applied in combination with the algorithm. Initially, we run the algorithm assuming that two rectangles are sufficient to completely learn the structure. In one pass, it is possible to verify that the solution generated does indeed correctly classify the input. If not, we can double the number of rectangles and try again. We continue doubling the number of rectangles until the algorithm successfully finds the nested rectangles, using $O(R)$ storage. Because the number of rectangles is doubled in each step, and successful classification is guaranteed when the number of rectangles is at least R, the partitioning algorithm will run at most $\lceil \log R \rceil$ times. Thus, the total number of passes needed to determine R and find the rectangles is $O(\log^{3/2} n \log R)$.

A generalization of the algorithm to higher dimensions is not obvious and is being investigated. We are considering a randomized variant of this algorithm for multiple dimensions. This variant partitions the input set into regions, each containing a fixed fraction of the points. Then the algorithm finds a rectangle in each region in each pass. While such an algorithm may work well in practice, it may not find the minimum set of rectangles, and may fail to find a correct generalization when one exists.

10. Lower bound

In this section we show that when the concept structure is fairly simple and the amount of storage necessary to represent the concept is therefore small, the number of passes required for *any* learning algorithm is proportional to the number of alternations. Intuitively, the alternations represent the concept boundaries, so the size of the representation constructed by most learning algorithms will be proportional to the number of alternations in the examples. For example, decision tree learning algorithms must construct one branch for every alternation to form a hypothesis consistent with the training set. When an algorithm is attempting to find a minimal-sized nested rectangle partitioning with R rectangles, it must find approximately $2dR$ alternations, where d is the number of attributes (dimensions) used to represent the examples. Similar results hold for other methods of partitioning into convex regions.

We will use a comparison-based model that is sufficiently strong to support our previous algorithms. The theorem below shows that any comparison-based algorithm that has $O(R)$ storage needs at least R passes through the examples when R is sufficiently small compared to the number of examples n.

First, we give an intuitive explanation using the case when $A = 4$ (that is, $R = 2$), where two passes are necessary to find the four alternations as long as the space available S satisfies $S < n/2$. We will show that any algorithm will make a mistake if it uses only one pass through a set of points that require four alternations. Suppose we have two categories, 'blue' and 'green'. Consider a trainer that presents points in the blue category first. Because the algorithm has limited storage, it will eventually forget at least one blue point. After all $n/2$ blue points have been presented, the trainer presents three points from the green category that surround the forgotten blue point. The trainer can choose either of two ways to position the green points (see Figure 3). In the figure, lower case letters represent examples, while upper case letters represent an induced convex partitioning of the line.

The algorithm can easily find the leftmost and rightmost alternations. To correctly locate the two inner alternations, though, it must surround the forgotten blue point. However, it cannot tell if the input is given by Case 1 or Case 2 of Figure 3. If the program were to decide one way, the trainer could have chosen to present the other case. Because the algorithm performed no comparisons between the forgotten blue points and the three green points, it cannot distinguish the two cases.

Theorem 10.1. *Consider a comparison-based algorithm that attempts to generate a correct representation of a 1-dimensional dataset having A alternations in P passes, using*

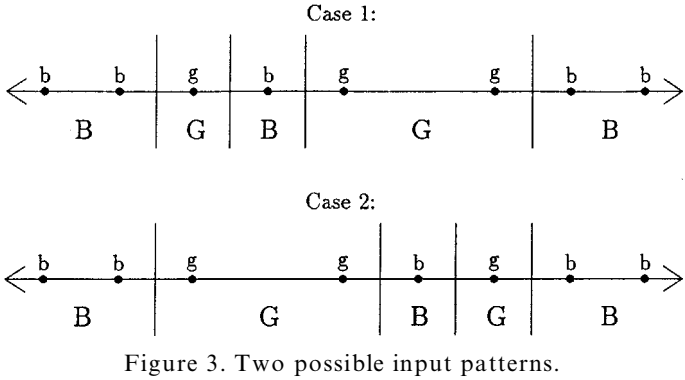

Figure 3. Two possible input patterns.

$S = O(A)$ storage. This algorithm may need $\lceil A/2 \rceil$ passes through the data set to find all A alternations when there are more than $2S(S+2)^P$ input points belonging to each class. This result holds even if the algorithm has seen some of the examples in a previous pass, as long as it has never compared two points belonging to different classes.

Proof. In this theorem, we define S to be the number of examples the algorithm can store in scratch memory, in addition to the training example under consideration. We prove by induction that for any algorithm we can construct two input configurations that are indistinguishable if $P < \lceil A/2 \rceil$. Our adversary will use the ability to change the order of the points presented in each pass. In particular, the adversary we choose will use two different orders on alternating passes. On odd passes, it will present all points in class C_1 before any in C_2. On even passes, it will present all points in class C_2 before any in C_1.

Base case. When $A = 1$ or $A = 2$, the theorem only requires that the algorithm make a single pass through the data. It is clear that at least one pass is required if no point in C_1 has been previously compared with any point in C_2.

Induction hypothesis. We claim the theorem holds for any number of alternations up to A.

Induction step. We now consider the case when there are $A+1$ alternations. We consider by way of contradiction an algorithm that can find all of the alternations in fewer than $\lceil (A+1)/2 \rceil$ passes. Let n_1 be the number of points in class C_1 and n_2 be the number of points in C_2. During the first pass, the adversary presents the points in class C_1. Since the algorithm can store no more than S of these points, there will be at least $\lceil n_1/S+1) - 1 \rceil$ adjacent points in C_1 which were forgotten by the algorithm. Therefore, during the remainder of the first pass, they cannot be compared to the points in category C_2 that will follow. Next, the adversary presents the n_2 points from class C_2. The adversary will put all of these points into the 'forgotten region' of C_1 points.

The algorithm will not be able to get any information about the C_2 points except for the fact that they are in the forgotten region. Therefore, while it may decide to find the boundary of the forgotten region, it will not be able to find any of the internal alternations. We are left with two sets of points: $\lceil n_1/(S+1) - 1 \rceil$ points in C_1 and n_2 points in C_2. They may merge into each other in any way—the algorithm has never compared any point in one set to a point in the other. Therefore, the alternations could occur anywhere in the forgotten region. So we have simply reduced the problem to the case of A, $A-1$, or $A-2$ alternations (depending on whether the algorithm takes advantage of the opportunity to find the two outermost alternations), with $\lceil n_1/(S+1) - 1 \rceil$ points remaining in C_1 and n_2 points in C_2, but now the algorithm has only $P-1$ passes. The next pass will be an even pass, so we can imagine swapping the points in C_1 and C_2 and using the same adversary algorithm. If, during this pass, the algorithm found any alternations, then we can apply the induction hypothesis directly with a smaller value for A. If not, we apply our induction step again (with appropriate changes to P, n_1, and n_2). By the induction hypothesis and the above argument, there are at least two different inputs that are indistinguishable by the current pass and the remaining passes, yet must be separated by different sets of rectangles. □

Note that since Theorem 10.1 forces the number of points in each class n to be greater than $2S(S+2)^P$ and ensures that $P \geqslant \lceil A/2 \rceil$, and since our 1-dimensional example can be embedded into any higher dimensional space, and since $S = O(A)$, we can restate the theorem in the following form:

Corollary 10.1. *Any comparison-based algorithm using $S = O(A)$ storage that generates a correct representation of a dataset having A alternations may require up to $\lceil A/2 \rceil$ passes to find all A alternations, when there are more than $2kA(A+2)^{\lceil A/2 \rceil}$ input points belonging to each class, for some constant k.*

11. Parallel algorithms

Connectionist learning architectures have natural parallel implementations. This has motivated us to examine whether our algorithms also have natural parallel implementations. We found that we can obtain optimal speed-ups given a few constraints; *e.g.* given n points on the line, we can solve the nested rectangle problem in $O(n/P)$ time using P processors, where $P \leqslant n/\log n$. We assume that input points $\{p_1, \ldots, p_n\}$ are stored in an array. We say that two points in the array are *adjacent* if their array indices differ by one. With P processors we can compare the category of each pair of adjacent points in $O(n/P)$ time. If the categories of p_i and p_{i+1} differ, then p_i and p_{i+1} define an alternation. For simplicity, assume the number of alternations is even; an extra alternation can be introduced at either end if necessary. To find the rectangles, we match up each alternation with another one that defines the opposite side of the same rectangle.

We accomplish the above by a simple application of parallel prefix sum operation (Karp and Ramachandran 1988) which is supported on the Connection Machine by the SCAN operation. Initially, we assign a value of 1 to each pair (p_i, p_{i+1}) that form an alternation, and a value of 0 to all other pairs. Applying a parallel-prefix sum operation will result in the ith alternation having value i, for $1 \leqslant i \leqslant 2R$, where R is the number of rectangles. The ith alternation can then write the midpoint of the two points that define the alternation into the ith location of an output array of size $2R$. The endpoints of rectangle i are located at positions i and $2R + 1 - i$ in the array. The entire computation (assuming the points are sorted initially) can be done optimally in $O(\log n)$ time with $O(n/\log n)$ processors or in $O(n/P)$ time with P processors when P is small. If the points are not sorted, the sorting step can be done in $O(\log^2 n)$ time, using the same number of processors (see Cole, 1988).

Parallel algorithms can be developed to solve the nested rectangle problem in any fixed number of dimensions. One would hope that our parallel algorithm could be generalized to work in any number of dimensions. However, we have shown that when the dimensionality is not fixed, the nested rectangle problem becomes log-space complete for P, or *P-complete* (see Appendix). This fact is usually interpreted to mean that no efficient (i.e. polylog time with polynomial number of processors) parallel algorithm exists, because there are no known methods to achieve significant speed-ups of such problems on parallel architectures.

12. Conclusions

Table 2 summarizes the upper and lower bounds discussed in this paper when the storage, S, is of size $O(R)$, where R is the number of rectangles needed to categorize the n input points. In the table, k and c are constants. Intuitively, our lower bound demonstrates that a learning program with fixed storage cannot create the optimal convex partitioning to classify a set of examples unless it is allowed to make many passes through the examples. The problem is that, since memory is limited, the program *must* forget some of the examples, and these examples may be misclassified

Table 2. Lower and upper bounds.

Number of rectangles	Lower bound	Upper bound
$kR(2R+2)^R < n$	R passes	R passes
$kR(2R+2)^R \geqslant n, R = O(\log^{3/2} n)$	None	R passes
$R \geqslant c \log^{3/2} n$	None	$O(\log^{3/2} n)$ passes

as a result. It is clear, however, that a partial generalization constructed in p passes can be further refined using more passes. This paper makes precise the trade-offs between storage, number of passes, and classification accuracy for a concept defined by nested hyperrectangles, which have been used experimentally in a number of systems.

When an algorithm only has enough memory to store the R rectangles themselves, it requires (in the worst case) no more than R passes through the data. In addition, when R is larger we can do much better. For instance, when $R \geqslant c \log^{3/2} n$, for some constant c, we only need $O(\log^{3/2} n)$ passes through the data. For most learning problems, we do not know the size of R in advance (i.e. we do not know how compact the generalization might be). However, our algorithm can still learn both R and the target concept accurately in $O(\log R \log^{3/2} n)$ passes.

From a practical standpoint, these results allow one to make some statements about the trade-off between processing time and storage for algorithms that learn nested hyperrectangles or other convex partitionings. As long as storage is not limited, we can use an algorithm that (with one pass through the data) runs in $O(dn \log n)$ time, where d is the number of features for each example, to find an optimal (i.e. minimum) set of rectangles. If storage is fixed and the number of examples is large, then $\min(R, O(\log^{3/2} n), n/S)$ passes are sufficient to find the optimal set of nested rectangles, depending on the size of R with respect to n. If fewer passes are allowed, then the concepts learned by the algorithm may misclassify some of the examples.

As a side note, we also have studied the complexity of non-incremental learning algorithms for parallel models of computation. We presented an optimal parallel algorithm for learning in one dimension, and have shown that learning nested hyperrectangles when the dimensionality is not fixed is P-complete, which implies that it is inherently sequential.

The major open problems we are currently considering include developing efficient limited memory algorithms for data sets with multiple dimensions. We plan to implement these algorithms for experimental tests on real data. We also are working on improvements to our currently crude parallel algorithms for multiple dimensions. Additionally, we are considering the problem of constructing approximately optimal concept structures.

To summarize, the main purpose of this paper was to increase awareness of the effect that limited memory has on a learning algorithm, in terms of both training time and accuracy. We studied this notion in the context of a particular model of learning where we were able to obtain precise quantitative trade-offs. While our study is primarily theoretical, and its intent is to increase our understanding of learning algorithms, this work is also interesting from the perspective of cognitive modelling. For instance, we conjecture that a heuristic version of one of the algorithms presented here may in fact be a plausible problem solving strategy for some perceptual learning tasks.

Acknowledgements

This research supported in part by Air Force Office of Scientific Research under Grant AFOSR-89-1151, National Science Foundation under Grants IRI-8809324, IRI-9116843, and IRI-9223591, and NSF/DARPA under Grant CCR-8808092. One of the authors (R.K.) is supported by NSF under Grant CCR-8804284 and NSF/DARPA under Grant CCR-8808092.

References

Aha, D. and Kibler, D. (1989) Noise-tolerant instance-based learning algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (Detroit, MI: Morgan Kaufmann) 794–799.

Carpenter, G. A., Grossberg, S. and Rosen, D. B. (1991) Fuzzy art: Fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, **4**(6), 759–771.

Christiensen, A. D. (1992) Learning to predict in uncertain continuous tasks. In *Machine Learning: Proceedings of the Ninth International Workshop* (San Mateo, CA: Morgan Kaufmann).

Cole, R. (1988) Parallel merge sort. *SIAM Journal on Computing*, **17**, 770–785.

Dobkin, D. and Munro, J. (1981) Optimal time minimal space selection algorithms. *Journal of the ACM*, **28**(3), 454–461.

Floyd, S. (1989) On Space-Bounded Learning and the Vapnik-Chervonenkis Dimension. PhD thesis, University of California at Berkeley, 1989.

Frievalds, R., Kinber, E. and Smith, C. (1993) On the impact of forgetting on learning machines. Technical Report UMIACS-TR-93-38, University of Maryland, College Park.

Haussler, D. (1988) Space-efficient learning algorithms. Technical Report UCSC-CRL-88-2, University of California at Santa Cruz, Department of Computer Science.

Heath, D., Kasif, S., Kosaraju, R., Salzberg, S. and Sullivan, G. (1990) Learning nested concept classes with limited storage. Technical Report JHU-90/9, Computer Science Department, The Johns Hopkins University.

Heath, D. (1992) A Geometric Framework for Machine Learning. PhD thesis, The Johns Hopkins University.

Helmbold, D., Sloan, R. and Warmuth, M. (1989) Learning nested differences of intersection closed concept classes. In *Proceedings of the 1989 Workshop on Computational Learning Theory*, San Mateo.

Karp, R. and Ramachandran, V. (1988) A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division, University of California, Berkeley.

Michalski, R. S. (1983) A theory and methodology of inductive learning. In R. Michalski, J. Carbonell and T. Mitchell (eds), *Machine Learning: An Artificial Intelligence Approach*, pp. 83–134 (San Mateo, CA: Morgan Kaufmann).

Minsky, M. and Papert, S. (1988) *Perceptrons*, (Cambridge, MA: MIT Press).

Munro, J. and Paterson, M. (1980) Selection and sorting with limited storage. *Theoretical Computer Science*, **12**(3), 315–323.

Pashler, A. and Badgio, P. (1985) Visual attention and stimulus identification. *Journal of Experimental Psychology: Human Perception and Performance*, 105–121.

Pylyshyn, Z. (1989) The role of location indexes in spatial perception: A sketch of the finst spatial index model. *Cognition*, **32**, 65–97.

Quinlan, J. R. (1986) Induction of decision trees. *Machine Learning*, **1**(1), 81–106.

Rosenblatt, F. (1959) Two theorems of statistical separability in the perceptron. In *Mechanisation of thought processes: Proceedings of a symposium held at the National Physical Laboratory, November*, pp. 421–456 (London: HM Stationery Office).

Salzberg, S. (1989) Nested hyper-rectangles for exemplar-based learning. In K. P. Jantke (ed.) *Analogical and Inductive Inference: International Workshop ATI '89*, (Berlin: Springer-Verlag), pp. 184–201.

Salzberg, S. (1990) *Learning with Nested Generalized Exemplars* (Norwell, MA: Kluwer Academic Publishers).

Salzberg, S. (1991) A nested hyperrectangle learning method. *Machine Learning*, **6**, 251–276.

Utgoff, P. (1989) Incremental induction of decision trees. *Machine Learning*, **4**(2), 161–186.

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

Appendix: Parallelizability

Theorem A.1. *The nested rectangle problem is P-complete.*

Proof. We have demonstrated that the nested rectangle problem is solvable in polynomial time. What remains to be shown is that it is log-space hard for P. We will do this by reducing a known P-complete problem, the monotone circuit value problem (MCVP), to the nested rectangle problem.

We are given an instance of MCVP as a list of n boolean formulas $\{g_1, \ldots, g_n\}$. Each formula g_i is either an *input* with the value 0 or 1, or a *gate* of the form $and(g_j, g_k)$, or $or(g_j, g_k)$, where $j, k > i$. We will use $\wedge$ for *and* and $\vee$ for *or*. We assume, without loss of generality, that g_{n-1} is the only input with value 0, and g_n is the only input with value 1. We can easily translate any instance of MCVP to this form if necessary. The problem is to calculate the value of g_1. We will translate this problem to a form of the nested rectangle problem in n dimensions. The particular form of nested rectangle problem we consider is 'Given a set of points from two classes, is it possible to correctly partition the points with any number of nested axis-parallel rectangles?' Such a partition is possible if and only if there does not exist a blocking rectangle. Each edge of a blocking rectangle is a hyperplane defined by $\{x \mid x_i = k_i\}$ for a dimension i and constant k_i. We call a hyperplane a *blocking hyperplane* if it is defined by $\{x \mid x_i = k_i\}$ and contains points of both classes. Thus, every edge of a blocking rectangle is a blocking hyperplane. We will use one dimension to store the value of each gate and input. The number of points we will generate is $2n+2+A+2O$, where A is the number of $\wedge$-gates, and O is the number of $\vee$-gates.

Initially, we start with the point 0 in class C_1 and the points at distance 2 and 4 along the positive halves of the coordinate axes in class C_2. This *initial configuration* contains $2n+1$ points. Figure 4 shows the initial configuration when $n = 2$. In the figure, solid circles represent points in class C_1, and hollow circles represent points in class C_2. Clearly, there are n blocking hyperplanes in the initial configuration, each of which contains the origin.

Next, we add the point $(0, 2, 2, \ldots, 2)$ in category C_1. This point has coordinate value 2 in all dimensions except the first, which is zero. There are exactly two blocking hyperplanes in every dimension but the first. For dimension $i, 1 < i \leqslant n$, the two blocking hyperplanes are $\{x \mid x_i = 0\}$ and $\{x \mid x_i = 2\}$. There is still only one blocking hyperplane, $\{x \mid x_1 = 0\}$, in the first dimension.

Assume temporarily that we had added $(2, 2, \ldots, 2)$, rather than $(0, 2, 2, \ldots, 2)$. Then the $2n$ blocking hyperplanes would define a unique blocking rectangle. We will add more points, so that a gate g_i has value 1 in the circuit if and only if there is a blocking rectangle containing the blocking hyperplane $\{x \mid x_i = 4\}$ as an edge under the above assumption. If there is a blocking rectangle R with blocking hyperplane $\{x \mid x_1 = 4\}$ as an edge, then R is still a blocking rectangle when the point $(2, 2, 2, \ldots, 2)$ is replaced with $(0, 2, 2, \ldots, 2)$ (*i.e.* we do not make the false assumption). Since we will add no point with its 1-coordinate not equal to 0 or 4 in the following construction, any blocking rectangle must have the hyperplane $\{x \mid x_1 = 4\}$ as an edge. Thus, a blocking rectangle exists if and only if the value of the circuit is 1.

We simulate $\wedge$-gates as follows: For $g_i = g_j \wedge g_k$, add a point, p, which has value 4

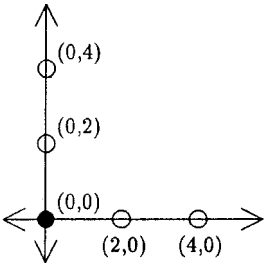

Figure 4. Initial set P in 2 dimensions.

in the i dimension, value 3 along dimensions j and k, and value 0 along all other dimensions. Assume there is a blocking rectangle with hyperplanes $\{x \mid x_j = 0\}$, $\{x \mid x_j = 4\}$, $\{x \mid x_k = 0\}$, and $\{x \mid x_k = 4\}$. Then this rectangle contains p and can contain the blocking hyperplane $\{x \mid x_i = 4\}$ as an edge. If either hyperplane $\{x \mid x_j = 4\}$ or $\{x \mid x_k = 4\}$ cannot be an edge of the blocking rectangle, then neither can hyperplane $\{x \mid x_i = 4\}$.

Now consider an $\vee$-gate, $g_i = g_j \vee g_k$. In this case, we add two points, p_1 and p_2, both with value 4 in the i dimension. One has value 3 in the j dimension; the other has value 3 in the k dimension. All other dimensions have value 0. If either hyperplane $\{x \mid x_j = 4\}$ or $\{x \mid x_k = 4\}$ were an edge of a blocking rectangle, then p_1 or p_2 would be contained in the rectangle, and hyperplane $\{x \mid x_i = 4\}$ could be an edge, as well. If neither is an edge of the rectangle, then hyperplane $\{x \mid x_i = 4\}$ cannot be.

To simulate g_{n-1}, which is the input with value 0, we add no points. Thus, the hyperplane $\{x \mid x_{n-1} = 4\}$ cannot be an edge of a blocking rectangle. For gate g_n, which is the input with value 1, we add the point $(1, 1, 1, \ldots, 1, 1, 4)$. It is always possible for the hyperplane $\{x \mid x_n = 4\}$ to be an edge of the blocking rectangle.

By the above discussion, we see a blocking hyperrectangle exists if and only if the associated MCVP problem has value 1. Thus, the problem of determining whether such a rectangle exists, and the problem of deciding whether a set of points can be classified by nested axis-parallel rectangles is P-complete. □