Perceptron-Based Learning Algorithms

STEPHEN I. GALLANT

***Abstract*—A key task for connectionist research is the development and analysis of learning algorithms. This paper examines several supervised learning algorithms for single-cell and network models. The heart of these algorithms is the pocket algorithm, a modification of perceptron learning that makes perceptron learning well behaved with nonseparable training data, even if that data is noisy and contradictory. Features of these algorithms include 1) *speed*—algorithms are fast enough to be able to handle large sets of training data; 2) *network scaling properties*—network methods scale up almost as well as single-cell models when the number of inputs is increased; 3) *analytic tractability*—upper bounds on classification error are derivable; 4) *on-line learning*—some variants can learn continually, without referring to previous data; and 5) *winner-take-all groups or choice groups*—algorithms can be adapted to select one out of *c* possible classifications. These learning algorithms are suitable for applications in machine learning, pattern recognition, and connectionist expert systems.**

I. Introduction

Connectionist models (or neural networks—the terms are used interchangeably here) have recently attracted large numbers of researchers from a variety of specialities [1]-[3]. Perhaps the key challenge of this field is the creation of good general-purpose learning algorithms.

This paper presents a review of several algorithms for supervised learning in discrete connectionist networks that are variants of the perceptron learning algorithm [4]. Much of this work has appeared previously but is scattered in conference proceedings [5]-[9]; collecting these techniques should make them more accessible for work in neural network learning, pattern recognition, and connectionist expert systems [10], [11]. While primarily a review paper, new simulations are given in Section III-A-4 and some new distribution free bounds are also given in Section VI. This paper is intended to be a sister paper to [10], with the focus on learning algorithms rather than inferencing and expert systems.

The main emphasis of this paper will be upon *functionality* rather than upon *modeling*. The goal is networks that can perform learning tasks as well as possible, regardless of "biological plausibility." In other words, there will be more concern with systems that produce correct responses than with systems that model human biases and errors.

The algorithms to be examined have several advantages over commonly used neural network techniques (such as back-propagation [12]-[14]), including:

- *Speed:* Fewer iterations are usually required and floating point computations can be avoided altogether. In fact, arithmetic operations can easily be limited to integer addition, subtraction, and comparisons.
- *Scaling properties:* Larger network models and more training examples can be handled.
- *Analytic tractability:* Analytic bounds on scaling and generalization can be derived in many cases, thanks to the simplicity of the underlying model.

Sections are organized as follows. Section II gives some basic definitions and details of the connectionist model. Single-cell algorithms are examined in Section III, and their generalizations to linear machine algorithms in Section IV. Several methods for constructing network models that have more than one cell are described in Section V, and some new generalization bounds are developed in Section VI. In order to cover many variations of these algorithms and for ease of reference, an attempt has been made to be concise so that sections read something like a reference catalog. For many of these topics more detail will be given in [15].

II. Basic Definitions

In supervised learning a set of training examples $\{E^k\}$ and corresponding correct responses $\{C^k\}$ is presented to a learning program, which must then model the underlying function. Each example E^k is a vector of p inputs: $\langle E_1^k, E_2^k, \cdots, E_p^k \rangle$. Except for the linear machine model of Section IV, each response C^k is a single scalar output (also called an activation). Multiple outputs (and autoassociative or content addressable memories) are easily handled by learning each output cell independently of other outputs.

Supervised learning should be contrasted with unsupervised learning; the latter assumes that no correct responses $\{C^k\}$ are provided to a learning program. While supervised learning can attempt to learn arbitrary functions, the most that can be expected with unsupervised learning is to cluster the data into similarity groupings under certain assumptions on the nature of the data.

On-line learning is also of interest. In this paradigm examples are presented one by one and the algorithm tries to produce a good (updated) model after each example without storing previously seen examples.

Manuscript received August 23, 1989; revised January 2, 1990. This work was supported in part by the National Science Foundation under Grant IRI-8611596.

The author is with the College of Computer Science, Northeastern University, Boston, MA 02115. Some of this work was performed at the Knowledge Systems Laboratory of the Nippon Telegraph and Telephone Corporation, Yokosuka, Japan.

IEEE Log Number 9034407.

1045-9227/90/0600-0179$01.00 © 1990 IEEE

Here the standard connectionist assumption will be made that cell i computes a single activation (i.e., local output) u_i, which may in turn serve as input to other cells and/or be considered as an output of the entire network. If u_i is restricted to a small finite set of values such as $\{+1, -1, 0\}$, then the model is *discrete*. Alternatively u_i might assume a value in some interval such as $[0, 1]$, in which case the model is *continuous*. An important example of a continuous model is provided by back-propagation networks [12]-[14]. Rumelhart and McClelland [1], Anderson and Rosenfeld [2], and Hinton [16] have surveyed learning algorithms for continuous connectionist networks.

Algorithms examined in this paper are based upon perhaps the simplest discrete model, one that has been referred to as a multilayer perceptron, a gamba perceptron [17], a linear discriminant network, or a threshold logic unit network. Here activations are $\{+1, -1, 0\}$ and cell i computes its output, u_i, by thresholding the weighted sum of its inputs, S_i:

$$S_i = w_{i,0} + \sum_j w_{i,j} u_j$$

$$u_i = \begin{cases} +1 & \text{if } S_i > 0 \\ -1 & \text{if } S_i < 0 \\ 0 & \text{if } S_i = 0. \end{cases}$$

The quantity $w_{i,0}$ is a constant added to the sum and is called the *bias*.

This paper will examine several algorithms that are variants of perceptron learning [4]. Other discrete models and learning algorithms have been studied by Nakano [18], Kohonen [19], Anderson [20], Amari [21], Steinbuch [22], and Hopfield [23]. These are "one shot" learning algorithms that look at every training example exactly one time. This makes learning very fast but limits the ability of these algorithms to fit arbitrary sets of training examples. In these models training example inputs must be approximately orthogonal or else performance degrades.

A. Continuous Values

Although the main focus is upon discrete models, it is important to note that a continuous value, x, can always be approximated to arbitrary precision by a collection of discrete activations $\{u_i\}$. For example a "thermometer code,"[1] where cells represent intervals of the form $x \geq c_i$, can be used. Thus a probability $x \in [0, 1]$ might be represented by

$$\begin{array}{lll} u_1 = +1 & \text{if } x \geq 0.2; & \text{otherwise } u_1 = -1 \\ u_2 = +1 & \text{if } x \geq 0.4; & \text{otherwise } u_2 = -1 \\ u_3 = +1 & \text{if } x \geq 0.6; & \text{otherwise } u_3 = -1 \\ u_4 = +1 & \text{if } x \geq 0.8; & \text{otherwise } u_4 = -1 \end{array}$$

so that $x = 0.55$ would be represented by $u_1 = +1$, $u_2 = +1$, $u_3 = -1$, and $u_4 = -1$. Other representations (such as base 2 encodings) are more compact but tend to be more difficult for learning algorithms to deal with.

Most of the following algorithms allow cell *inputs* to be continuous values, even though the cell *output* is discrete. For example the pocket algorithm (described below) works for rational valued inputs. If training examples are limited in number and inputs are more naturally represented as continuous values, then it may be better to leave inputs continuous rather than "Booleanizing" them with a thermometer code. Having fewer inputs requires fewer weights in the model and therefore helps avoid overfitting a limited amount of training data.

III. Single-Cell Models

Networks are assumed to have p inputs, each taking on values in $\{+1, -1, 0\}$, and a single (output) cell with index $p + 1$. For all algorithms there is a standard trick for computing the bias, $w_{p+1,0}$. Each training example is merely augmented with an extra 0th input that always takes on the value $+1$. This lets the bias weight $w_{p+1,0}$ be generated just like the other p weights, $w_{p+1,i}$, using methods that assume a bias of 0.

The key algorithm to be discussed is the pocket algorithm [5], a modification of perceptron learning [4], [17], [24], [25]. Perceptron learning (defined in Section III-A-5 below) is well suited for *separable problems*, i.e., problems for which there exists some set of weights $\{w_{p+1,i}\}$ that correctly classifies all training examples. For separable problems perceptron learning will find some set of weights that correctly classifies all training examples after a finite number of mistakes. The algorithm also works for infinite sets of training examples with real-valued inputs if 1) lengths of training examples are bounded, i.e., $\|E^k\| < L$ for some L and 2) there exists a set of weights, $w_{p+1,*}$, and $\delta > 0$ such that all training examples E^k satisfy

$$\sum_j w_{p+1,j} E_j^k \geq \delta \qquad \text{for } C_k = +1$$

and

$$\sum_j w_{p+1,j} E_j^k \leq -\delta \qquad \text{for } C_k = -1.$$

See Minsky and Papert for details [17].

More recently Littlestone [26] has developed the Winnow algorithm, which is faster than perceptron learning for separable data when the number of inputs, p, is large and many of these inputs are not relevant. However if the number of such irrelevant inputs is not large, then Winnow can be slower than perceptron learning. Winnow also requires a problem-specific parameter to be set by the user.

Nonseparable problems are a different story. Since no set of weights can correctly classify all training examples, the best that can be hoped for is a set of weights that correctly classifies as large a fraction of the training examples as possible. Such a set of weights is called *optimal*.

[1] The term is attributed to B. Widrow of Stanford University.

Note that there are alternatives that do not fit the training data as well, for example computing weights that give minimum squared error. Such alternatives are necessary for algorithms, such as back-propagation, that require a differentiable error function.

Perceptron learning is not well behaved for nonseparable problems. While it will eventually visit an optimal sets of weights, it will not converge to *any* set of weights. Even worse, the algorithm can go from an optimal set of weights to a worst-possible set in one iteration, regardless of how many iterations have been taken previously (see Table II below). The pocket algorithm makes perceptron learning well behaved by adding positive feedback in order to stabilize the algorithm. The Appendix contains a more precise definition and proof of such stability.

The following subsections present the pocket algorithm and give several variants for different classes of problems. Each subsection will present an algorithm and then describe when that algorithm is most appropriate, give theoretical results and generalization bounds, and conclude with comments.

A. The Pocket Algorithm with Ratchet

1) Applicability: Finite set of training examples. Examples may be repeated, noisy, and contradictory ($E^k = E^l$, $C^k \neq C^l$).

2) Algorithm: The basic idea of perceptron learning is to take a training example, E^k, that is incorrectly classified by the current set of weights and to add E^k to the current weights if $C^k = 1$ or subtract E^k from the current weights if $C^k = -1$.

The basic idea of the pocket algorithm is to run perceptron learning while keeping an extra set of weights "in your pocket." Whenever the perceptron weights have a longest run of consecutive correct classifications of randomly selected training examples, these perceptron weights replace the pocket weights. The pocket weights are the outputs of the algorithm (see Fig. 1).

As an example of perceptron learning and the pocket algorithm, consider the XOR problem with training examples

$$E^1 = \langle +1 \quad -1 \quad -1 \rangle \qquad C^1 = -1$$
$$E^2 = \langle +1 \quad -1 \quad +1 \rangle \qquad C^2 = +1$$
$$E^3 = \langle +1 \quad +1 \quad -1 \rangle \qquad C^3 = +1$$
$$E^4 = \langle +1 \quad +1 \quad +1 \rangle \qquad C^4 = -1.$$

(The first entry for each training example is +1 for computing the bias weights.)

```
INPUT: Training Examples { $E^k, C^k$ }. $E^k$ is a vector with $E_0^k \equiv 1$ and other
    components, $E^k{}_1, \ldots, E^k{}_p$, assuming values in $\{+1, -1, 0\}$. $C^k = \{+1, -1\}$
    is the desired response.

OUTPUT: $W = <w_{p+1,0}, w_{p+1,1}, \cdots, w_{p+1,p}>$ is a vector of integral 'pocket'
    weights where $w_{p+1,0}$ is the bias.

TEMPORARY DATA:
    $\pi$ = vector of integral perceptron weights, $<\pi_0, \pi_1, \cdots, \pi_p>$.
    run$_\pi$ = number of consecutive correct classifications using perceptron
        weights $\pi$.
    run$_W$ = number of consecutive correct classifications using pocket weights
        $W$.
    num_ok$_\pi$ = total number of training examples that $\pi$ correctly classifies.
    num_ok$_W$ = total number of training examples that $W$ correctly classifies.

1.  Set $\pi = <0, 0, \cdots, 0>$ and
    run$_\pi$ = run$_W$ = num_ok$_\pi$ = num_ok$_W$ = 0.
2.  Randomly pick a training example $E^k$ (with corresponding clas-
    sification $C^k$).
3.  If $\pi$ correctly classifies $E^k$, i.e.
                $\{\pi \cdot E^k > 0$ and $C^k = +1\}$ or
                $\{\pi \cdot E^k < 0$ and $C^k = -1\}$
    Then:
        3a.  run$_\pi$ = run$_\pi$ + 1.
        3b.  If run$_\pi$ > run$_W$
             Then:
                3ba.  Compute num_ok$_\pi$ by checking every
                      training example.
                3bb.  (Ratchet:) If num_ok$_\pi$ > num_ok$_W$
                      Then:
                        3bba.  Set $W = \pi$
                        3bbb.  Set run$_W$ = run$_\pi$
                        3bbc.  Set num_ok$_W$ = num_ok$_\pi$
                        3bbd.  If all training
                               examples are correctly classi-
                               fied (i.e. num_ok$_W$ = $|\{ E^k \}|$)
                               then stop; the training exam-
                               ples are separable.
    Otherwise:
        3A.  (Change step:) Form a new vector of perceptron
             weights
                    $\pi = \pi + C^k E^k$
        3B.  Set run$_W$ = 0
4.  End of this iteration. If the specified number of iterations has
    not been taken then go to 2.
```

Fig. 1. Pocket algorithm with ratchet. Perceptron weights, π, are computed that occasionally replace pocket weights, W.

Iter.	π	run$_\pi$	W	run$_W$	Choice	OK?	Action
1.	< 0 0 0>	0	< 0 0 0>	0	E^4	no	$\pi = \pi - E^4$ Run$_\pi$ = 0
2.	<−1 −1 −1>	0	< 0 0 0>	0	E^4	yes	Run$_\pi$ = Run$_\pi$ + 1 $W = \pi$ Run$_W$ = Run$_\pi$
3.	<−1 −1 −1>	1	<−1 −1 −1>	1	E^2	no	$\pi = \pi + E^2$ Run$_\pi$ = 0
4.	< 0 −2 0>	0	<−1 −1 −1>	1	E^3	no	$\pi = \pi + E^3$ Run$_\pi$ = 0
5.	< 1 −1 −1>	0	<−1 −1 −1>	1	E^4	yes	Run$_\pi$ = Run$_\pi$ + 1
6.	< 1 −1 −1>	1	<−1 −1 −1>	1	E^2	yes	Run$_\pi$ = Run$_\pi$ + 1 $W = \pi$ Run$_W$ = Run$_\pi$
7.	< 1 −1 −1>	2	< 1 −1 −1>	2	E^3	yes	Run$_\pi$ = Run$_\pi$ + 1 Run$_W$ = Run$_\pi$
8.	< 1 −1 −1>	3	< 1 −1 −1>	3	E^1	no	$\pi = \pi - E^1$ Run$_\pi$ = 0
9.	< 0 0 0>	0	< 1 −1 −1>	3	⋯		

Fig. 2. Pocket algorithm iterations.

Fig. 2 gives a typical sequence of iterations. At iteration 8 both pocket and perceptron weights are optimal, classifying three out of four training examples correctly. But at iteration 9 the initial perceptron weights, $\langle 0 \quad 0 \quad 0 \rangle$, have been reached. These misclassify every training example whereas the pocket weights, $\langle 1 \quad -1 \quad -1 \rangle$, still get three out of four correct. As iterations continue, changes to the pocket weights will become less and less frequent. The time between such changes increases exponentially with respect to Run$_W$. Most of these changes will replace one set of optimal weights by another; for example $\langle 1 \quad 1 \quad 1 \rangle$ is another set of optimal weights. From time to time, however, nonoptimal weights will appear in the pocket, but the next theorem states that this will happen less and less frequently as the number of iterations increases.

3) Theory:

Theorem 1: Given a finite set of input vectors $\{E^k\}$ and corresponding desired responses $\{C^k\}$ and a probability $P < 1$, there exists an N such that after $n \geq N$ iterations of the pocket algorithm, the probability that the pocket coefficients are optimal exceeds P.

Proof: See the Appendix.

4) Generalization Bounds and Simulations: (see Table III for distribution-free generalization bounds): Previous experiments [5] have indicated that the pocket algorithm is better able to fit a set of training examples than a standard statistical method, Wilks method, as implemented in SPSS-X [27]. Experiments indicated that roughly 20% fewer errors were made on the training data.

To give some idea of generalization differences, three new experiments were run with results summarized in Table I. The first used weather data of Osaka to predict if Tokyo would receive rain the next day.[2] The second set consisted of (proprietary) financial data.[3] The final test was parity-5, training only, using all 32 examples.[4]

Note in Table I the higher classification rates on both training and testing data for the pocket algorithm with ratchet compared with perceptron learning. The standard deviations are given in parenthesis and suggest greater stability for the former algorithm. To give some idea of speed, 30 000 iterations of the parity-5 problem consume about 30 s on a Sun 4.

Also note that for large sets of training examples the bounds in Table III indicate that generalization percentages will be close to training percentages, so that being able to fit the training data is most important.

5) Comments:

- If branches 3a and 3b are removed, then the resulting algorithm is perceptron learning (with output vector π rather than W). Note that the change step (3A) merely adds or subtracts the current training example to the perceptron weights depending upon whether the correct response, C^k, is $+1$ or -1.
- If the training examples are separable, then only a finite number of mistakes (i.e., executions of the change step, 3A) can be made before pocket weights are optimal. This is the perceptron convergence theorem [4], [17].
- The ratchet check, step 3bb, ensures that whenever pocket coefficients W are changed (in step 3bba) they are strictly better than the previous weights. Thus the quality of the weights ``ratchets up.''

TABLE I
COMPARISON OF PERCEPTRON LEARNING AND POCKET ALGORITHM WITH RATCHET: AVERAGES AND STANDARD DEVIATIONS ARE COMPUTED OVER TEN TRIALS EACH

	Perceptron		**Pocket+Ratchet**	
	Train %	**Test %**	**Train %**	**Test %**
Weather Data	71.3 %	67.0 %	80.9 %	76.6 %
	(11.0)	(10.0)	(0.5)	(1.8)
Financial Data	54.4 %	51.3 %	62.5 %	63.6 %
	(3.6)	(7.5)	(1.0)	(1.6)
Parity-5 Data	43.0 %	—	65.8 %	—
	(5.9)	—	(6.5)	—

- There is no good way of specifying how many iterations to run; in fact, the problem of deciding whether data are separable or nonseparable is not easily bounded [25] unless polynomial linear programming techniques are employed. However, the following procedure may be helpful. Start with $i = 10\,000$ iterations. If weights are changed (step 3bba) for any iteration after the first $i/5$ iterations, then at the end of this set of iterations run another set of i additional iterations with i now increased to $1.5i$. Continue in this manner until a set of iterations is run with no weight replacement during the last $0.8i$ iterations. For example a weight replacement at iteration number 3800 would trigger 15 000 more iterations after the first 10 000 were completed. If there were a weight replacement at iteration 4500 of the new set, then an additional 22 500 iterations would be run after the second set of 15 000 was completed.
- The number of iterations required to produce an *optimal* set of weights is prohibitively large for most problems. However, experience [5], [9], including the Table I simulations above, indicates that *good* weights are produced using reasonable amounts of computation time.
- Yoshida *et al.* [28] created a medical connectionist expert system using the pocket algorithm and compared its classification performance with a standard statistical packet, SAS 'DISCRIM' [29]. They found improved testing (71.8% versus 63.2%) and training (91% versus 67%) performance for the pocket algorithm. There were 334 training examples and 163 test examples, each with nine continuous inputs (represented as 27 discrete values) and four possible disease classification outputs.
- Mooney *et al.* [30] compared standard perceptron learning, back-propagation, and Quinlan's ID3 [31] (a decision tree method) and reported that perceptron learning performance was ``hardly distinguishable'' on four of the five tests they ran and that on the fifth test it did ``about as well as ID3'' (but not as well as back-propagation with additional intermediate cells). Training times were reported as one to two orders of magnitude faster than back-propagation.

[2] Every three consecutive days for a year were randomly divided into two days for training and one for testing, giving 247 total training examples and 127 total test examples. Each example had eight real-valued inputs, each of which was normalized and rounded to integers in the range [−100, +100]. The output was Boolean. Fifty thousand iterations were performed for each algorithm using ten different random seeds (to change the order of selection of training examples). Data courtesy of R. Nakano and K. Saito of Knowledge Systems Lab, NTT Information Processing Labs, Yokosuka, Japan.

[3] There were 1564 training examples and 814 test examples, each with 29 Boolean inputs. Again 50 000 iterations were performed for each algorithm using ten different random seeds.

[4] Thirty thousand iterations were performed for each algorithm. Optimal performance is 69%, which the pocket algorithm with ratchet achieved on eight of ten trials.

INPUT: Training Examples $\{ E^k, C^k \}$. E^k is a vector with $E_0^k \equiv 1$ and other components, ${E^k}_1, \ldots, {E^k}_p$, assuming values in $\{+1, -1, 0\}$. $C^k = \{+1, -1\}$ is the desired response. Examples do not reside in memory but are produced one-by-one (sampling with replacement) whenever needed.

OUTPUT: $W = <w_{p+1,0}, w_{p+1,1}, \cdots, w_{p+1,p}>$ is a vector of integral 'pocket' weights where $w_{p+1,0}$ is the bias.

TEMPORARY DATA:

π = vector of integral perceptron weights, $<\pi_0, \pi_1, \cdots, \pi_p>$.
$\mathbf{run}_\pi$ = number of consecutive correct classifications using perceptron weights π.
$\mathbf{run}_W$ = number of consecutive correct classifications using pocket weights W.

```
1.  Set $\pi = <0, 0, \cdots, 0>$ and
    $run_\pi = run_W = num\_ok_\pi = num\_ok_W = 0$.
2.  Obtain a training example $E^k$ (with corresponding classification $C^k$).
3.  If $\pi$ correctly classifies $E^k$, i.e.
                $\{\pi \cdot E^k > 0$ and $C^k = +1\}$ or
                $\{\pi \cdot E^k < 0$ and $C^k = -1\}$
    Then:
        3a.  $run_\pi = run_\pi + 1$.
        3b.  If $run_\pi > run_W$
             Then:
                3ba.  Set $W = \pi$
                3bb.  Set $run_W = run_\pi$
    Otherwise:
        3A.  (Change step:) Form a new vector of perceptron weights
                $\pi = \pi + C^k E^k$
        3B.  Set $run_W = 0$
4.  End of this iteration. If the specified number of iterations has
    not been taken then go to 2.
```

Fig. 3. Pocket algorithm for ∞ training data. Training examples are not stored.

B. The Pocket Algorithm For ∞ Training Data

1) Applicability: Training data that are not simultaneously available in storage due to on-line learning or simulated noise. This is referred to as ∞ *training data*.

2) Algorithm: This algorithm is the same as the pocket algorithm with ratchet, except that no ratchet check can be performed because not all examples are held in storage (see Fig. 3).

3) Theory and Generalization Bounds: Same as Sections III-A-3 and III-A-4.

4) Simulated Noise: Sometimes a process can be modeled by a relatively small set of noise-free training examples and a probabilistic noise model; yet generating an exhaustive set of noisy training examples would require too much storage. There are two approaches for handling such problems. One is to generate a large but manageable set of noisy training examples and use the pocket algorithm with ratchet. The other is to use the pocket algorithm for ∞ training data and to dynamically generate training examples as needed.

Using the ratchet in the first approach has the advantage of producing a better fitting model, but for a smaller sample of data. The second approach allows more training examples to be used and reduces storage requirements. Both give a simple way to test resulting coefficients against new data: merely turn off the learning for a batch of new test data. Such testing gives closer estimates than the distribution-free bounds of Section VI and takes advantage of the inexhaustible supply of data.

INPUT: Training Examples $\{ E^k, C^k \}$. E^k is a vector with $E_0^k \equiv 1$ and other components, ${E^k}_1, \ldots, {E^k}_p$, assuming values in $\{+1, -1, 0\}$. $C^k = \{+1, -1\}$ is the desired response. The first r training examples are *rules* that must not be violated by the pocket weights.

OUTPUT: $W = <w_{p+1,0}, w_{p+1,1}, \cdots, w_{p+1,p}>$ is a vector of integral 'pocket' weights where $w_{p+1,0}$ is the bias.

TEMPORARY DATA:

π = vector of integral perceptron weights, $<\pi_0, \pi_1, \cdots, \pi_p>$.
$\mathbf{run}_\pi$ = number of consecutive correct classifications using perceptron weights π.
$\mathbf{run}_W$ = number of consecutive correct classifications using pocket weights W.
$\mathbf{num\_ok}_\pi$ = total number of training examples that π correctly classifies.
$\mathbf{num\_ok}_W$ = total number of training examples that W correctly classifies.

```
1.  Set $\pi = <0, 0, \cdots, 0>$ and
    $run_\pi = run_W = num\_ok_\pi = num\_ok_W = 0$.
2.  Let $\pi$ be the current perceptron weights. If $\pi$ does not satisfy
    all $r$ rules, then pick $E^k$ to be any rule that is misclassified; otherwise
    select $E^k, k > r$, at random from the remaining training examples.
    Let $C^k$ be the corresponding correct classification of $E^k$.
3.  If $\pi$ correctly classifies $E^k$, i.e.
                $\{\pi \cdot E^k > 0$ and $C^k = +1\}$ or
                $\{\pi \cdot E^k < 0$ and $C^k = -1\}$
    Then:
        3a.  $run_\pi = run_\pi + 1$.
        3b.  If $run_\pi > run_W$
             Then:
                3ba.  Compute $num\_ok_\pi$ by checking every training example.
                3bb.  (Ratchet:) If $num\_ok_\pi > num\_ok_W$
                      Then:
                         3bba.  Set $W = \pi$
                         3bbb.  Set $run_W = run_\pi$
                         3bbc.  Set $num\_ok_W = num\_ok_\pi$
                         3bbd.  If all training examples are correctly classified
                                (i.e. $num\_ok_W = |\{ E^k \}|$) then stop;
                                the training examples are separable.
    Otherwise:
        3A.  (Change step:) Form a new vector of perceptron weights
                $\pi = \pi + C^k E^k$
        3B.  Set $run_W = 0$
4.  End of this iteration. If the specified number of iterations has
    not been taken then go to 2.
```

Fig. 4. Modification to the pocket algorithm to accommodate rules.

C. The Pocket Algorithm with Rules and Examples

1) Applicability: The first r examples, $\{E^1, E^2, \cdots, E^r\}$, are considered to be *rules* that must not be violated by the pocket weights. The goal is weights that correctly classify as many training examples as possible while making no mistakes on any of the r rules. It is assumed that the rules constitute a separable set of training examples.

2) Algorithm: The pocket algorithm is modified so that after the perceptron weights change, only misclassified rules are selected for iterations until all rules become correctly classified (see Fig. 4).

3) Theory: The pocket convergence theorem is easily extended to accommodate rules.

4) Comments:

- The rule modification can be used with the ratchet or the ∞ example version of the pocket algorithm. The change to step 2 prevents any set of invalid perceptron weights from ever becoming pocket weights.
- For the finite training example case, another way to implement rules is to make several copies of the r rules and add them to the set of examples before ap-

plying the pocket algorithm with ratchet. If enough copies are added it will become too expensive for a set of weights to misclassify any of these rules.

IV. Linear Machines

Linear machines, sometimes called winner-take-all groups or choice groups, are credited to Nilsson [24], [25], [32]. They are generalizations of single-cell models where several cells compete to determine which one of them will *fire* (output = +1) while all remaining cells *do not fire* (output = −1). Fig. 5 illustrates the situation. The c cells, numbered $p + 1, p + 2, \cdots, p + c$, compute weighted sums

$$S_i = \sum_{j \geq 0} w_{i,j} u_j$$

as before, but the single cell that fires is determined by the first cell with the highest weighted sum:

$$u_i = \begin{cases} +1 & \text{if} \begin{cases} S_i > S_j & \text{for } i > j \text{ and} \\ S_i \geq S_j & \text{for } i < j \end{cases} \\ -1 & \text{otherwise.} \end{cases}$$

Training example outputs, C^k, are each c vectors consisting of $c - 1$ components of -1 and one component of $+1$.

Linear machines are very handy for pattern recognition, fault detection, and any other problem where input vectors must be classified into one of c categories. It is well worth emphasizing that, despite their name, linear machines do not compute linear functions of their inputs.

The following facts are either well known or easily proved:

1) Single-cell models can be represented as 2-cell linear machines.
2) A 2-cell linear machine can be represented by (and hence is equivalent to) a single-cell model [24], [25].
3) For $c > 2$, the output patterns from a c-cell linear machine need not be representable by c independent single-cell models. For example the last cell in Fig. 5 computes the XOR function. It can be shown that the final cell in a linear machine with at most 2^p cells can be made to compute *any* Boolean function (without adding intermediate cells).
4) For any linear machine and any cell i where $(p + 1 \leq i \leq p + c)$ there exists an equivalent linear machine where cell i has all weights and bias equal to 0 (i.e., $w_{i,j} = 0$ for $j = 0, 1, \cdots, p$). (For example cell 3 in the figure has all 0 weights.)
5) The problem of learning weights for a linear machine with c cells and p inputs using T training examples can be converted to a problem of learning weights for a single-cell model with cp inputs and cT training examples. This is due to a construction by Kessler [25].

The next task is to extend the single-cell learning algorithms to linear machines.

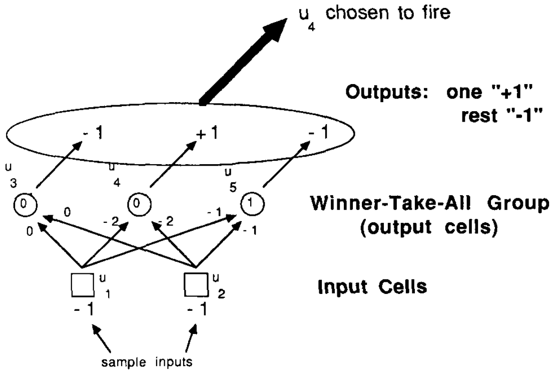


Fig. 5. Linear machine. The numbers inside the circles are biases.

A. *The Pocket Algorithm with Ratchet for Linear Machines*

1) Applicability: Classifying input vectors into one of c possible groups. The network model is generated from a finite set of training examples. Examples may be repeated, noisy, and contradictory.

2) Algorithm: Two modifications are needed to apply the pocket algorithm to linear machines. First a perceptron weight change consists of adding the current example to the weights for the desired output cell and subtracting the current example from the weights of exactly one of the other cells that had at least as large a weighted sum. The second modification is that for pocket weight changes the weights from all c perceptron cells are put into the pocket as a group (see Fig. 6).

3) Theory: The pocket convergence theorem generalizes to linear machines in the obvious way. The key argument is that the perceptron convergence and perceptron cycling theorems generalize to linear machines because of fact 5) above. The proof is omitted.

4) Generalization Bounds: See Table III for distribution-free generalization bounds.

5) Comments:

- As with single-cell models, the algorithm usually will not produce *optimal* weights for nonseparable problems unless prohibitively many iterations are run. Nevertheless in practice the algorithm can be expected to produce *good* weights with reasonable computational effort. As an example, tests reported in [9] consistently gave at least 85% of the optimal performance.
- The linear machine model with c cells generally gives better results than training c independent single-cell models because exactly one of the c cells is guaranteed to be selected.

B. *The Pocket Algorithm for Linear Machines with ∞ Training Data*

1) Applicability: Classifying input vectors into one of c possible groups. Training data are not simultaneously available in storage due to on-line learning or simulated noise.

```
INPUT: Training Examples { $E^k, C^k$ }. $E^k$ is a vector with $E_0^k \equiv 1$ and other
    components, $E^k{}_1, \ldots, E^k{}_p$, assuming values in $\{+1, -1, 0\}$. $C^k$ is an output
    vector with $c$ entries, $C^k{}_{p+1}, \cdots, C^k{}_{p+c}$, one of which is +1 and the other
    $c - 1$ taking value $-1$.

OUTPUT: $c$ cells $W_{p+1}, W_{p+2}, \cdots, W_{p+c}$ where $W_i = <w_{i,0}, w_{i,1}, \cdots, w_{i,p}>$ is
    a vector of integral 'pocket' weights and $w_{i,0}$ is the bias.

TEMPORARY DATA:
    $\pi$ = $c$ vectors of integral perceptron weights, $\pi_{p+1}, \pi_{p+2}, \cdots, \pi_{p+c}$, where
        each vector $\pi_i = <\pi_{i,0}, \pi_{i,1}, \cdots, \pi_{i,p}>$ .
    $\text{run}_\pi$ = number of consecutive correct classifications using perceptron
        weights $\pi$.
    $\text{run}_W$ = number of consecutive correct classifications using pocket weights
        $W$.
    $\text{num\_ok}_\pi$ = total number of training examples that $\pi$ correctly classifies.
    $\text{num\_ok}_W$ = total number of training examples that $W$ correctly classifies.

1. Set $\pi_i = <0, 0, \cdots, 0>$ for $i = p+1, \ldots, p+c$;
   $\text{run}_\pi = \text{run}_W = \text{num\_ok}_\pi = \text{num\_ok}_W = 0$.
2. Randomly pick a training example $E^k$ (with corresponding classification $C^k$).
3. If $\pi$ correctly classifies $E^k$, i.e. if the correct class is $i$ and
           $\{\pi_i \cdot E^k > \pi_j \cdot E^k$ for $i \neq j\}$
   Then:
       3a. $\text{run}_\pi = \text{run}_\pi + 1$.
       3b. If $\text{run}_\pi > \text{run}_W$
           Then:
               3ba. Compute $\text{num\_ok}_\pi$ by checking every
                    training example.
               3bb. (Ratchet:) If $\text{num\_ok}_\pi > \text{num\_ok}_W$
                    Then:
                        3bba. Set $W = \pi$. Note that all $c$
                              of the vectors are replaced at
                              once.
                        3bbb. Set $\text{run}_W = \text{run}_\pi$
                        3bbc. Set $\text{num\_ok}_W = \text{num\_ok}_\pi$
                        3bbd. If all training
                              examples are correctly classi-
                              fied (i.e. $\text{num\_ok}_W = |\{ E^k \}|$)
                              then stop; the training exam-
                              ples are separable.
   Otherwise:
       3A. (Change step:) Form a new vector of perceptron
           weights. If $C^k{}_i = +1$ then select one cell $j \neq i$ that
           satisfies $\pi_i \cdot E^k \leq \pi_j \cdot E^k$ and modify the weights of
           cells $i$ and $j$ as follows:
                   $\pi_i = \pi_i + E^k$
                   $\pi_j = \pi_j - E^k$
       3B. Set $\text{run}_W = 0$
4. End of this iteration. If the specified number of iterations has
   not been taken then go to 2.
```

Fig. 6. The pocket algorithm with ratchet for linear machines.

2) Algorithm: See Fig. 7.

3) Theory: The pocket convergence theorem holds as before. There is also an interesting relationship between linear machines and optimal Bayesian decision rules for noisy pattern recognition and fault-detection problems. Consider the problem of trying to classify a pattern of p Boolean inputs into one of c classes under the following restrictions:

1) Each class has some fixed (prior) probability of occurrence.
2) In the absence of noise each class corresponds to a *single* pattern among the p inputs. Thus there would be only c possible input patterns if noise were not present.
3) Inputs are independently corrupted by noise according to a fixed set of error probabilities $\{P_{ij}\}$, with each $P_{ij} < 1/2$. If the correct class for a pattern is i, then the probability its jth feature will be inverted is P_{ij}. (Every input feature E_j^k is ± 1.) Note that if all $P_{ij} > 0$ then any input pattern might correspond to any class (perhaps with very low probability).

```
INPUT: Training Examples { $E^k, C^k$ }. $E^k$ is a vector with $E_0^k \equiv 1$ and other
    components, $E^k{}_1, \ldots, E^k{}_p$, assuming values in $\{+1, -1, 0\}$. $C^k$ is an output
    vector with $c$ entries, $C^k{}_{p+1}, \cdots, C^k{}_{p+c}$, one of which is +1 and the other
    $c - 1$ taking value $-1$. Examples do not reside in memory but are produced
    one-by-one (sampling with replacement) whenever needed.

OUTPUT: $c$ cells $W_{p+1}, W_{p+2}, \cdots, W_{p+c}$ where $W_i = <w_{i,0}, w_{i,1}, \cdots, w_{i,p}>$ is
    a vector of integral 'pocket' weights and $w_{i,0}$ is the bias.

TEMPORARY DATA:
    $\pi$ = $c$ vectors of integral perceptron weights, $\pi_{p+1}, \pi_{p+2}, \cdots, \pi_{p+c}$, where
        each vector $\pi_i = <\pi_{i,0}, \pi_{i,1}, \cdots, \pi_{i,p}>$ .
    $\text{run}_\pi$ = number of consecutive correct classifications using perceptron
        weights $\pi$.
    $\text{run}_W$ = number of consecutive correct classifications using pocket weights
        $W$.

1. Set $\pi_i = <0, 0, \cdots, 0>$ for $i = p+1, \ldots, p+c$;
   $\text{run}_\pi = \text{run}_W = \text{num\_ok}_\pi = \text{num\_ok}_W = 0$.
2. Obtain a training example $E^k$ (with corresponding classification $C^k$).
3. If $\pi$ correctly classifies $E^k$, i.e. if the correct class is $i$ and
           $\{\pi_i \cdot E^k > \pi_j \cdot E^k$ for $i \neq j\}$
   Then:
       3a. $\text{run}_\pi = \text{run}_\pi + 1$.
       3b. If $\text{run}_\pi > \text{run}_W$
           Then:
               3ba. Set $W = \pi$. Note that all $c$ of the vectors
                    are replaced at once.
               3bb. Set $\text{run}_W = \text{run}_\pi$
   Otherwise:
       3A. (Change step:) Form a new vector of perceptron
           weights. If $C^k{}_i = +1$ then select one cell $j \neq i$ that
           satisfies $\pi_i \cdot E^k \leq \pi_j \cdot E^k$ and modify the weights of
           cells $i$ and $j$ as follows:
                   $\pi_i = \pi_i + E^k$
                   $\pi_j = \pi_j - E^k$
       3B. Set $\text{run}_W = 0$
4. End of this iteration. If the specified number of iterations has
   not been taken then go to 2.
```

Fig. 7. The pocket algorithm for linear machines with ∞ training data.

The task is to determine the *most likely* classification based upon a noisy input pattern.

It is well known [32], [25] that the optimal Bayesian decision rule for this problem can be expressed as a linear machine (see also [9]). Furthermore it is possible (in theory) to represent all possible noisy inputs and correct classes by a finite set of training examples that faithfully preserves their respective probabilities. (Of course this is not practical due to the astronomical number of training examples that would be required.) This argument shows that if the pocket algorithm is presented with noisy training examples along with their correct classifications, then the weights the algorithm produces will converge (in probability) to weights that implement the optimal Bayesian decision rule in a linear machine. It is also possible to incorporate prior probabilities and costs by adjusting the frequency according to which training examples are chosen. For more on this topic see [8] and [9].

V. Network Models

This section examines the problem of data-fitting with a network rather than a single-cell model. Of course the only reason to use a network is to fit the training examples—and hopefully the actual data—more closely than would be possible with a single cell.

Two assumptions are made. First it is assumed there is no prespecified network topology, so that any network is acceptable if it performs the learning or pattern recognition task.[5]

It is also assumed that a certain continuity or "robustness preserving" property is desirable:

> *An unseen input* E *that is very close (in Hamming distance) to a training example* E^k *usually will produce the same output as* E^k *from the network.*

The qualification "usually" serves to avoid an overly strict application of this principle that would rule out everything except nearest neighbor algorithms.

The following algorithms are motivated by an "engineering approach": decompose the problem and use previously developed algorithms wherever possible. In fact these two algorithms reduce the network learning problem to a single-cell problem (with close to single-cell speed). Both approaches work with any of the variants of the pocket algorithm (including the linear machine version) of Sections III and Section IV.

The *distributed method* uses a single layer of d intermediate cells with fixed randomly generated weights as in Fig. 8. Note that the single top cell sees both the original inputs and the activations from the intermediate cells. The intermediate cells create a distributed representation [33] in a higher dimensional space. Thus each example will have $p + d$ rather than p features. As Cover [34] has shown, this makes a set of training examples more likely to be separable and hence easier to learn.

The second approach is the *tower algorithm*, illustrated in Fig. 9. Initially a single-cell model is generated, these weights are frozen, and then a new trainable cell is added that sees the original inputs plus the activations from the cell immediately below. Continuing in this fashion the output from each level improves upon that of the previous level under conditions discussed in Section V-B below.

A. Distributed Method

1) Applicability: Fitting finite or ∞ training data better than is possible with a single-cell model. Examples may be repeated, noisy, and contradictory.

2) Algorithm: See Fig. 10.

3) Theory and Generalization Bounds: See Section VI.

4) Comments:

- Experience has shown the distributed method to be a much quicker way to fit data than back-propagation; the speed ratio is typically one to three orders of magnitude. Learning speed is critical for real-world problems that are data rich, because a faster algorithm can fit larger sets of training data and the amount of training data modeled can be the most important consideration when fitting data. Thus learning speed affects generalization for data-rich problems(!).

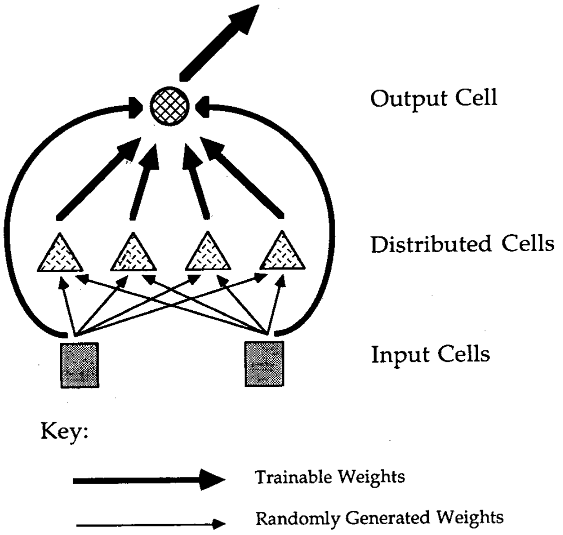


Fig. 8. The distributed method.

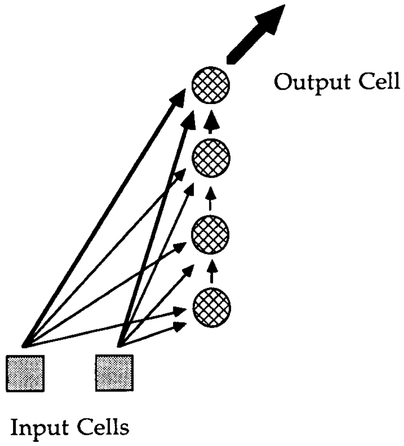


Fig. 9. The tower algorithm.

1. Generate a layer of d intermediate cells just above the input layer as in figure 8. Each intermediate cell sees all p input cells and has integer weights and bias generated at random in some range $[-K, K]$. Once generated these distributed cell weights do not change.
2. Remaining cells in the network use single-cell or linear machine learning algorithms. Note, however, that their inputs now include the outputs from the newly added distributed cells.

Fig. 10. Algorithm for the distributed method.

Recently this viewpoint has been supported in two pattern recognition studies [35], [36]. Hayashi *et al.* obtained over 99.5% generalization rates on multifont character recognition using the distributed method. Large amounts of training data were modeled: 20 000 examples, each with 272 inputs and 60 to 90 outputs. They reported that it would not have been practical to use back-propagation with such

[5] The back-propagation algorithm for continuous network models is more appropriate for the harder problems where network topology is prespecified.

large amounts of data.[6] Three of their main findings were:

1) The more training examples the better the generalization.
2) A suitable number of iterations for this problem was about ten times the number of training examples.
3) For 16 × 16 input grids (plus some additional features) a suitable number of distributed cells was about $\frac{1}{2}p$, i.e., half the total number of input cells.

- On the other hand if a prespecified network topology is required or if the number of cells in the network must be minimized, then back-propagation would be a better choice than the distributed method.[7]
- The distributed method is "robustness preserving" by the following argument. Let E and E' be two inputs that are close (i.e., $\|E - E'\|/p$ is small). Then a distributed cell i will tend to produce the same output for E and E' because their weighted sums, S_i^E and $S_i^{E'}$, will tend to agree in sign. (This argument was formalized by Amari [37], [38].) As for output cells in the network, their input patterns for E and E' will also be close (since most of the distributed cell outputs will agree). Thus their activations will also tend to agree for E and E', establishing the robustness-preserving property of the network.
- Types of cells other than threshold logic units with randomly generated coefficients can be used for distributed cells, provided such cells help separate sets of inputs while tending to agree on similar input sets. Radial basis functions used by Renals and Rohwer [39] provide one example. Another example is Kawahara's [40] use of clustering algorithms on the examples to generate intermediate cells.
- Precomputing the activations for all distributed cells is an important computational speedup for finite sets of training examples.
- It can be advantageous to make several tries to fit data using different randomly generated weights for distributed cells. See Section VI and [41].

B. *The Tower Algorithm*

The tower algorithm employs single-cell learning to build a tower of cells, where each cell sees the original inputs and the single cell immediately below. Fig. 11 gives the algorithm and Fig. 12 shows the algorithm applied to the parity-5 problem. (This computation took well under 1 minute on a Sun 3.)

1) Applicability: Fitting finite or ∞ training data better than is possible with a single-cell model. Examples may be repeated, noisy, and contradictory.

1. Use the pocket algorithm to generate a single-cell model and freeze these weights.
2. Create a new cell that sees the p inputs to the network and the activation from the cell that was most recently trained. Run the pocket algorithm to train the $p+2$ weights (including bias) for this cell.
3. If the network with this added cell gives improved performance, then freeze its coefficients and go to step 2; otherwise remove this last added cell and output that network.

Fig. 11. The tower algorithm.

```
.                          bias  in1  in2  in3  in4  in5  tv1  tv2
Tower Var. 1 for parity:    0    -2   -2   -2   -2    2    0    0
Tower Var. 2 for parity:    1     3    3    3    3   -3    6    0
parity (output):            0    -2   -2   -2   -4    2    0    9
```

Fig. 12. Sample solution for parity-5 with two intermediate cells found by the tower algorithm.

2) Algorithm: See Fig. 11.

3) Theory:

Theorem 2: For noncontradictory sets of training examples with input values restricted to $\{+1, -1\}$, the tower algorithm will fit the data with arbitrarily high probability, provided enough iterations are taken at each step.

Proof: This is easy to see because an n-cell model that misclassifies example E^k can be made into an $n + 1$ cell model that correctly classifies E^k and leaves other classifications unchanged. To show this, let $C^k = \pm 1$ be the correct classification of E^k. Now set

$$w_{n+1,j} = C^k E_j^k \qquad \text{for } j = 1, \cdots, p$$

$$w_{n+1,n} = p$$

$$w_{n+1,0} = C^k.$$

For example Fig. 13 gives a cell that copies the classification of the cell immediately below, except that $E^k = \langle -1, +1, +1, -1 \rangle$ now produces output $C^k = -1$.

The pocket algorithm converges on an optimal set of weights; therefore such an optimal set must correctly classify at least one more training example than the previous n-cell network because such a network has been constructed. This establishes the claimed theoretical convergence of the algorithm. □

4) Comments:

- In practice the tower algorithm can get bogged down when presented with large amounts of data because not enough iterations can be made to ensure an optimal solution at each step. Also there is no easy formula for how many iterations to run at each level. For these reasons the distributed method seems preferable for creating networks from data. Also the distributed method tends to be faster than the tower algorithm.
- The construction in Fig. 12 suggests how parity p can be computed with $n = [(p + 1)/2]$ intermediate and output cells and $n(p + 1) + n - 1$ weights. The algorithm discovered this fact(!); the author had thought that p intermediate and output cells would be

[6] Personal communication.

[7] The fact that back-propagation uses continuous rather than discrete activations is glossed over here. However, weights produced by back-propagation can be used in discrete models and weights produced by the distributed method, pocket algorithm, etc., can be used in some continuous models.

required for the parity p problem until he tested the algorithm.

- Linear machines can be used at each stage in place of single-cell models.

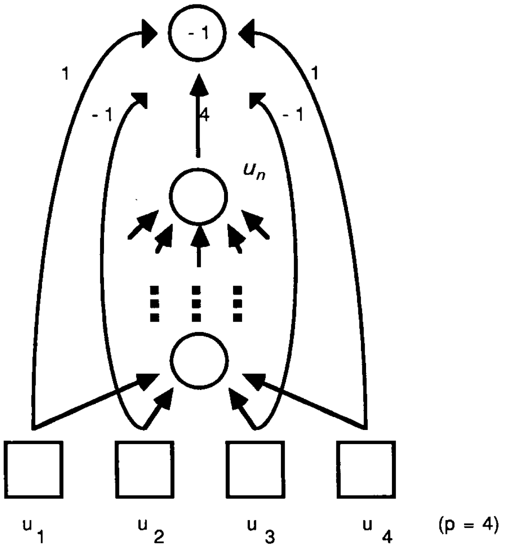


Fig. 13. The added tower cell classifies $E^k = \langle -1, +1, +1, -1 \rangle$ as false (-1) and copies u_n for other inputs.

VI. Analytic Bounds

A previous paper [41] analyzed a generalization of the distributed method called the BRD (bounded, randomized, distributed) Algorithm. This algorithm differs from the distributed method by making up to T tries to fit data with distributed networks rather than only 1 try. By specializing the BRD bounds it is possible to compute generalization bounds for the algorithms that were presented in previous sections.

A. Notation and Theorem

Notation is given in Table II.

Theorem 3 (Gallant [41]): Let T distributed networks, each with d intermediate cells, be generated. Suppose one of these networks with top cell length L misclassifies a fraction $\epsilon^0 > 0$ of the E training examples. For any $\epsilon > \epsilon^0$ let $s = (\epsilon - \epsilon^0)/\epsilon$ be the slack between ϵ and the measured error ϵ^0. If the number of training examples E is larger than the minimum of

$$\frac{8}{s^2\epsilon} \max\left[\ln\frac{8}{\delta}, \min\left\{2(p + Td + 1),\right.\right.$$
$$\left.\left. 4(pd + 2d + p + 1)\log\left(e(d+1)\right)\right\} \ln\frac{16}{s^2\epsilon}\right]$$

and

$$\frac{\ln\frac{1}{\delta} + (d + p + 1)\ln\left(T(2L+1)\right)}{s^2\epsilon} \min\left\{\frac{1}{2\epsilon}, 2\right\}$$

then with confidence at least $1 - \delta$ that network will have actual error at most ϵ.

TABLE II
Notation for Analytic Bounds

E	Number of training examples in the training set. Training examples are assumed to be selected one-by-one with replacement.
ε	Actual error of a network. ε gives the probability that a sample drawn from the total collection of data will be misclassified. Data may be noisy and may contain repeated and contradictory examples.
ε°	Measured error on the training set of E examples.
s	Slack between measured error and actual error, $s = \frac{\varepsilon - \varepsilon^\circ}{\varepsilon}$
$1 - \delta$	Confidence that the model will fit with error at most ε.
T	Number of tries using the distributed method to fit the data. ($T = 1$ for the distributed method on a single-cell model as previously described.)
p	Number of input cells.
d	Number of distributed cells in the intermediate layer of cells. $d = 0$ for a single-cell model.
c	Number of output cells for linear machines or number of intermediate and output cells for the tower algorithm. $c = 1$ for single-cell models.
L	Length of the weight vector for the top cell $L = \|W\| = \sqrt{\sum_{j=0}^{p+d} w_{p+d+1,j}^2}$
log	Logarithm base 2.
ln	Logarithm base e.
e	Base of the natural logarithm.

If some model correctly classifies every training example ($\epsilon^0 = 0$) then the conclusion holds if

$$E \geq \frac{1}{-\ln(1-\epsilon)}\left[\ln T + \left\{(d + p + 1)\right.\right.$$
$$\left.\left.\cdot \ln(2L + 1)\right\} + \ln\left(\frac{1}{\delta}\right)\right].$$

B. Bounds

From Theorem 3 the required number of training examples, E, can be derived for a given confidence ($1 - \delta$) and error ϵ as follows:

1) For single-cell models (pocket algorithm with ratchet or ∞ training data or rules) specialize Theorem 3.
2) For linear machines with c output cells and p input cells it is known by Kessler's construction [25] that there exists an equivalent single-cell problem with $c(p + 1)$ input cells and bias of 0. The single-cell model can be further simplified by subtracting the first $p + 1$ weights from each of the sets of c weights leaving a single-cell model with at most $(c - 1)(p + 1)$ adjustable weights and 0 bias. This allows substitution into the single-cell bound with $p' = (c - 1)(p + 1) - 1$ to derive generalization bounds for linear machines.
3) For the distributed method use Theorem 3 with $T = 1$.
4) For the tower algorithm a general network bound due to Baum and Haussler [42] is used. This bound also gives part of the bound of Theorem 3.

TABLE III
BOUNDS FOR VARIOUS ALGORITHMS

Model and Number of examples, E, required for confidence $(1-\delta)$ and error ε where $s = (\varepsilon - \varepsilon^\circ)/\varepsilon$

Single-Cell Models:
pocket algorithm with ratchet
pocket algorithm for ∞ training data
pocket algorithm with rules

Minimum of:

$$\frac{8}{s^2\varepsilon}\max\left[\ln\frac{8}{\delta},\ \min\{2(p+1),\ 4(p+1)\log e\}\ln\frac{16}{s^2\varepsilon}\right]$$

or

$$\frac{\ln\frac{1}{\delta}+(p+1)\ln(2L+1)}{s^2\varepsilon}\min\left\{\frac{1}{2\varepsilon},2\right\}$$

Linear Machine Models

Minimum of:

$$\frac{8}{s^2\varepsilon}\max\left[\ln\frac{8}{\delta},\ \min\{2(c-1)(p+1),\ 4(c-1)(p+1)\log e\}\ln\frac{16}{s^2\varepsilon}\right]$$

or

$$\frac{\ln\frac{1}{\delta}+(c-1)(p+1)\ln(2L+1)}{s^2\varepsilon}\min\left\{\frac{1}{2\varepsilon},2\right\}$$

Distributed Method

Minimum of:

$$\frac{8}{s^2\varepsilon}\max\left[\ln\frac{8}{\delta},\ \min\{2(p+Td+1),\ 4(pd+2d+p+1)\log(e(d+1))\}\ln\frac{16}{s^2\varepsilon}\right]$$

or

$$\frac{\ln\frac{1}{\delta}+(d+p+1)\ln(T(2L+1))}{s^2\varepsilon}\min\left\{\frac{1}{2\varepsilon},2\right\}$$

Tower Algorithm

$$\frac{8}{s^2\varepsilon}\max\left[\ln\frac{8}{\delta},\ 4(c(p+2)-1)\log(ec)\ln\frac{16}{s^2\varepsilon}\right]$$

TABLE IV
SOME SAMPLE DISTRIBUTION-FREE BOUNDS ON THE NUMBER OF TRAINING EXAMPLES REQUIRED TO ENSURE GENERALIZATION

Model	ε	ε°	s	δ	p	L	c	d	T	Examples
single-cell	.20	.10	.50	.9	10	30	1	0	1	1813
linear machine	.20	.10	.50	.9	10	100	3	0	1	4671
distributed	.20	.10	.50	.9	10	50	1	5	10	4432
tower	.20	.10	.50	.9	10	0	4	0	1	597346
single-cell	.15	.10	.33	.9	10	30	1	0	1	5439
linear machine	.15	.10	.33	.9	10	100	3	0	1	14013
distributed	.15	.10	.33	.9	10	50	1	5	10	13295
tower	.15	.10	.33	.9	10	0	4	0	1	2133342
single-cell	.10	.05	.50	.9	10	30	1	0	1	3626
linear machine	.10	.05	.50	.9	10	100	3	0	1	9342
distributed	.10	.05	.50	.9	10	50	1	5	10	8863
tower	.10	.05	.50	.9	10	0	4	0	1	1338251

C. *Comments*

As an example of how these numbers scale, the bounds for several typical cases are computed in Table IV. Note that distribution-free bounds are very conservative and that alternatives have been proposed, for example by Buntine [43]. Also note that the bound for the tower algorithm appears very loose and that a significantly lower distribution-free bound may be possible.

VII. DISCUSSION

A number of simple, fast, and analytically tractable variants of the pocket algorithm modification to perceptron learning have been examined. Their advantage over pure perceptron learning is their ability to handle nonseparable, noisy, and contradictory data.

It is important to try to characterize those problems that are more suitable for perceptron-based learning algorithms and those that are more suitable for the widely used back-propagation algorithm. Because of speed, the perceptron-based algorithms seem preferable for data-rich, large-scale problems that are essentially discrete in nature. These algorithms are also recommended for generating networks to be used in connectionist expert systems as described in [10], because inferencing and rule generation are much simpler without the back-propagation sigmoid function.

By contrast if a problem constrains the network topology (number of cells and their connections), then back-propagation is the only choice for generating weights. Back-propagation also seems preferable for problems with mostly continuous data, particularly if the network *outputs* are continuous.

It is difficult to compare the algorithms for the case where training data are limited and where generalization to unseen data is of primary importance.

In conclusion it is worth noting that a number of researchers are experimenting with perceptron-based algorithms and creating still other pocket algorithm variants. For example Buntine [44] has added a "pocket" to Littlestone's Winnow algorithm, Utgoff [45] has experimented with combining the pocket algorithm with decision trees, and Mézard and Nadal [46] have looked at network generation methods that use the pocket algorithm. It seems likely that such algorithms will play an important role in machine learning, connectionist expert systems, and pattern recognition applications.

APPENDIX
PROOF OF THE POCKET CONVERGENCE THEOREM

Theorem 1: Given a finite set of input vectors $\{E^k\}$ and corresponding desired responses $\{C^k\}$ and a probabiltiy $p < 1$, there exists an N such that after $n \geq N$ iterations of the pocket algorithm, the probability that the pocket coefficients are optimal exceeds p.

The proof is in several steps:

1) *Only a finite number of sets of coefficients can be reached using perceptron learning.* This is a restatement of the perceptron cycling theorem [17], [47].

2) *From any such set of coefficients there is a nonzero probability that perceptron learning will visit an optimal set of coefficients (perhaps after several steps).*

Let π be the current set of coefficients. Choose any optimal set of coefficients and the subset, S, of training examples for which it produces correct responses. By the perceptron convergence theorem [4], [17] if inputs from S are repeatedly chosen, in finite time perceptron learning will go from π to π' where π' are coefficients that give desired responses for all inputs in S. Thus π' must be an optimal set of coefficients for all of $\{E^k\}$.

Since there is a nonzero probability that this exact sequence of vectors in S may be chosen when vectors from E are chosen randomly, an optimal solution can be reached from π with nonzero probability.

3) *As the number of iterations grows, perceptron learning will visit optimal sets of coefficients a nonzero fraction of the time.*

This follows directly from 1) and 2). Fig. 14 which illustrates coefficient sets may be useful; arrows indicate possible successor sets of coefficients from step (3A) of the pocket algorithm and asterisks indicate optimal sets. Notice the optimal sets (E, F, and G) may be visited less often than other sets (for example B).

4) *Let set ν be a nonoptimal set of coefficients that gives correct results for randomly selected inputs with probability P_ν and let Ω be an optimal set of coefficients giving correct results with probability P_Ω ($> P_\nu$), and suppose ν is visited M times for every time Ω is visited, where M is fixed and may be greater than 1. Then as the number of iterations n grows, the probability that*

> *{the longest run of consecutive correct responses with Ω is greater than the longest run with ν}*

approaches 1.

Proof: The probability that the longest run for ν is $< k$ for N trials is easily seen to be

$$(1 - P_\nu^k)^N \tag{1}$$

and similarly for Ω.

It suffices to show that given $0 < \sigma < 1$, there is an N and k satisfying:

(a) Prob {longest run for ν is $< k$ for (MN) trials} $> \sigma$

(b) Prob {longest run for Ω is $\geq k$ for N trials} $> \sigma$.

Let N be fixed. From (1), condition (a) is satisfied for

$$k \geq \frac{\log\left(1 - \sqrt[MN]{\sigma}\right)}{\log P_\nu}. \tag{2}$$

For condition (b) it is desired that

$$1 - (1 - P_\Omega^k)^N > \sigma$$

or

$$k < \frac{\log\left(1 - \sqrt[N]{(1 - \sigma)}\right)}{\log P_\Omega} \tag{3}$$

since $\log P_\Omega < 0$. Thus it must be shown that for large enough N

$$\frac{\log\left(1 - \sqrt[MN]{\sigma}\right)}{\log\left(1 - \sqrt[N]{(1 - \sigma)}\right)} < \frac{\log P_\nu}{\log P_\Omega}. \tag{4}$$

Since $0 < P_\nu < P_\Omega < 1$ (because Ω optimal)

$$\log P_\nu < \log P_\Omega < 0$$

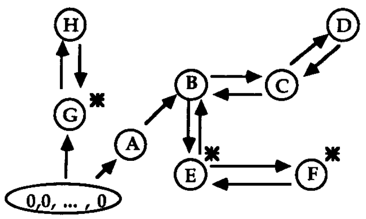


Fig. 14. Learning coefficients visited in perceptron learning.

so

$$1 < \frac{\log P_\nu}{\log P_\Omega}.$$

Thus letting $C_1 = \sqrt[M]{\sigma}$ *and* $C_2 = (1 - \sigma)$, it suffices to show that

$$\lim_{N \to \infty} \frac{\log\left(1 - \sqrt[N]{C_1}\right)}{\log\left(1 - \sqrt[N]{C_2}\right)} = 1.$$

This follows by two applications of l'Hôpital's rule.

5) *Steps 1, 3, and 4 imply the theorem.* If the problem is separable, the pocket algorithm will produce an optimal set of coefficients by the perceptron convergence theorem. Otherwise, perceptron learning will go from coefficient set to coefficient set, but eventually one of the repeatedly visited optimal sets will have a longer run of correct responses than any other particular nonoptimal set with arbitrarily high probability. Since there are only a finite number of nonoptimal sets of coefficients, as the number of iterations grows an optimal set will, with probability 1, have the longest run of correct responses. □

Acknowledgment

The author wishes to thank the reviewers for helping to improve this paper.

References

[1] D. E. Rumelhart and J. L. McClelland, Eds., *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1. Cambridge, MA: MIT Press, 1986.

[2] J. A. Anderson and E. Rosenfeld, Eds., *Neurocomputing, A Reader*. Cambridge, MA: MIT Press, 1988.

[3] S. Grossberg, *Neural Networks and Natural Intelligence*. Cambridge, MA: MIT Press, 1988.

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

[5] S. I. Gallant, "Optimal linear discriminants," in *Proc. Eighth Int. Conf. Pattern Recognition* (Paris, France), Oct. 28-31, 1986, pp. 849-852.

[6] S. I. Gallant, "Three constructive algorithms for network learning," in *Proc. Eighth Ann. Conf. Cognitive Sci. Soc.* (Amherst, MA), Aug. 15-17, 1986, pp. 652-660.

[7] S. I. Gallant and D. Smith, "Random cells: An idea whose time has come and gone . . . and come gain?" in *Proc. IEEE Int. Conf. Neural Networks* (San Diego, CA), vol. II, June 1987, pp. 671-678.

[8] S. I. Gallant, "Automated generation of expert systems for problems involving noise and redundancy, in *Proc. AAAI Workshop Uncertainty in Artificial Intelligence* (Seattle, WA), July 10-12, 1987, pp. 212-221.

[9] S. I. Gallant, "Bayesian assessment of a connectionist model for fault detection," in *Proc. AAAI Fourth Workshop Uncertainty in Artificial Intelligence* (St. Paul, MN), Aug. 19-21, 1988, pp. 127-135.

[10] S. I. Gallant, "Connectionist Expert Systems," *Commun. ACM*, vol. 31, no. 2, pp. 152–169, Feb. 1988. (Japanese translation in *Neurocomputer*, published by Nikkei Artificial Intelligence, pp. 114–136, 1988.)

[11] S. I. Gallant, "Matrix controlled expert system producible from examples," U.S. Patent 4 730 259, Mar. 8, 1988.

[12] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard University, 1974.

[13] D. B. Parker, "Learning logic," Tech. Rep. TR-47, Center for Computational Research in Economics and Management Science, MIT, Apr. 1985.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, D. E. Rumelhart and J. L. McClelland, Eds. vol. 1. Cambridge, MA: MIT Press, 1986.

[15] S. I. Gallant, *Connectionist Learning and Expert Systems*. Cambridge, MA: MIT Press (in preparation).

[16] G. E. Hinton, "Connectionist learning procedures," Tech. Rep. CMU-CS-87-115, Department of Computer Science, Carnegie-Mellon University, 1984; also *Artificial Intellig.*, to be published.

[17] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969.

[18] K. Nakano, "Associatron—A model of associative memory," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-2, pp. 380–388, July 1972.

[19] T. Kohonen, "Correlation matrix memories," *IEEE Trans. Comput.*, vol. C-21, pp. 353–359, 1972.

[20] J. A. Anderson, J. W. Silverstein, S. A. Ritz, and R. S. Jones, "Distinctive features, categorical perception, and probability learning: Some applications of a neural model," *Psych. Rev.*, vol. 84, pp. 413–451, 1977.

[21] S. Amari, "Learning patterns and pattern sequences by self-organizing nets of threshold elements," *IEEE Trans. Comput.*, vol. C-21, pp. 1197–1206, Nov. 1972.

[22] K. Steinbuch, "Die Lernmatrix," *Kybernetik*, vol. 1, pp. 36–45, 1961.

[23] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat. Acad. Sci. U.S.*, vol. 79, pp. 2554–2558, 1982.

[24] N. J. Nilsson, *Learning Machines*. New York, NY: McGraw-Hill, 1965.

[25] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. New York: Wiley, 1973.

[26] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Machine Learning*, vol. 2, pp. 285–318, 1988.

[27] *SPSS-X User's Guide*. New York: McGraw-Hill, 1984.

[28] K. Yoshida, Y. Hayashi, and A. Imura, "A connectionist expert system for diagnosing hepatobiliary disorders," in *MEDINFO '89 Proc. Sixth Conf. Med. Informatics* (Beijing), 16–20 Oct. 1989 and (Singapore), 11–15 Dec. 1989. Amsterdam: North-Holland, pp. 116–120.

[29] *SAS User's Guide: Statistics*, pp. 381–396, 1982.

[30] R. Mooney, J. Shavlik, G. Towell, and A. Gove, "An experimental comparison of symbolic and connectionist learning algorithms," in *Proc. AAAI '89* (Detroit, MI), pp. 775–780.

[31] J. R. Quinlan, "Learning efficient classification procedures and their application to chess end games," in *Machine Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Palo Alto, CA, Tioga, 1983.

[32] M. Minsky and O. G. Selfridge, "Learning in random nets" (Fourth Symp. Inform. Theory, London, England, 1960), in *Information Theory*, C. Cherry, Ed. Washington: Butterworths, 1961.

[33] G. E. Hinton, "Distributed representations," Tech. Rep. CMU-CS-84-157, Department of Computer Science, Carnegie-Mellon University; revised version in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, vol. 1, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, 1986.

[34] T. M. Cover, "Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition," *IEEE Trans. Electron. Comput.*, vol. 14, pp. 326–334, 1965.

[35] Y. Hayashi, M. Sakata, T. Nakao, and S. Ohhashi, "Alphanumeric character recognition using a connectionist model with the pocket algorithm," *Int. J. Neural Networks: Res. & Appl.*, vol. 1, no. 3, pp. 175–186, 1989.

[36] H. Y. Lee, Y. C. Lee, and H. H. Chen, "Hand written letter recognition with neural networks," in *Proc. Int. Joint Conf. Neural Networks* (Washington, DC), June 18–22, 1989, p. II-618.

[37] S. A. Amari, "Method of statistical neurodynamics," *Kybernetik*, vol. 14, pp. 201–215, (Heft 4) Apr. 1974.

[38] S. Amari, "Mathematical foundations of neurocomputing," Rep. METR89-06, Faculty of Engineering, University of Tokyo, August 1989. (Journal version to appear.)

[39] S. Renals and R. Rohwer, "Phoneme classification experiments using radial basis functions," in *Proc. Int. Joint Conf. Neural Networks* (Washington, DC), June 18–22, 1989, pp. I-461–I-467.

[40] H. Kawahara and T. Irino, "Introduction to saturated projection algorithm for artificial neural network design," Res. Rep., NTT Basic Research Laboratories, Musashino, Tokyo, Japan. (English summary of Japanese IEICE reports PRU88-54 and SP88-86, 1988.)

[41] S. I. Gallant, "A connectionist learning algorithm with provable generalization and scaling bounds," *Neural Networks*, vol. 3, no. 2, 1990.

[42] E. B. Baum and D. Haussler, "What size net gives valid generalization?" *Neural Computation*, vol. 1, no. 1, pp. 151–160, 1989.

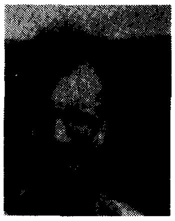
[43] W. Buntine, "A critique of the valiant model," in *Proc. AAAI '89* (Detroit, MI), pp. 837–842.

[44] W. Buntine, "Inductive knowledge acquisition and induction methodologies," *Knowledge-Based Systems*, vol. 2, no. 1, pp. 52–61, 1989.

[45] P. E. Utgoff, "Perceptron trees: A case study in hybrid concept representations," in *Proc. Seventh Nat. Conf. Artificial Intelligence*, Aug. 1988, pp. 601–606.

[46] M. Mézard and J.-P. Nadal, "Learning in feedforward layered networks: The tiling algorithm," *J. Phys. A: Math. & Gen.*, vol. 22, no. 12, pp. 2191–2203, 1989.

[47] H. D. Block and S. A. Levin, "On the boundedness of an iterative procedure for solving a system of linear inequalities," in *Proc. AMS*, 1970, pp. 229–235.



Stephen I. Gallant received the B.S. degree in mathematics from the Massachusetts Institute of Technology, Cambridge, MA, and the Ph.D. degree in operations research from Stanford University, Stanford, CA.

He is currently an Associate Professor of Computer Science at Northeastern University, Boston, MA. His research centers on machine learning in connectionist (or neural network) models with applications to connectionist expert systems. His book *Connectionist Learning and Expert Systems* is to be published by MIT Press.