Minimum Generalization via Reflection: A Fast Linear Threshold Learner

Steven Hampson and Dennis Kibler
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

hampson@ics.uci.edu, kibler@ics.uci.edu

September 9, 1998

Abstract

The number of adjustments required to learn the average LTU function of d features, each of which can take on n equally spaced values, grows as approximately n^{2d} when the standard perceptron training algorithm is used on the complete input space of n^d points and perfect classification is required. We demonstrate a simple modification that reduces the observed growth rate in the number of adjustments to approximately $d^2(log(d) + log(n))$ with most, but not all input presentation orders. A similar speed-up is also produced by applying the simple but computationally expensive heuristic "don't overgeneralize" to the standard training algorithm. This performance is very close to the theoretical optimum for learning LTU functions by any method, and is evidence that perceptron-like learning algorithms can learn arbitrary LTU functions in polynomial, rather than exponential time under normal training conditions. Similar modifications can be applied to the Winnow algorithm, achieving similar performance improvements and demonstrating the generality of the approach.

Keywords: LTU, Winnow, Perceptron, Reflection, Generalization

Running Head: Reflection

1 Introduction

A Linear Threshold Unit (LTU) defines a $(d-1)$-dimensional hyperplane that divides a d-dimensional space into two regions, and given a linearly separable sets of points, a simple training algorithm exists that is guaranteed to find such a separating hyperplane. This is arguably the simplest classification function and is a highly abstract model of the computational and learning capabilities of a single neuron. As such it has received a great deal of attention over a considerable period of time (Nilsson, 1965; Muroga, 1971; Minsky & Papert, 1969; Duda & Hart, 1973). Somewhat surprisingly though, many of the basic properties of LTU representation and training are still not known precisely. This includes the number of LTU functions, maximum and average function complexity as measured by required weight size, and training time as measured by the number of mistakes/adjustments made before a correctly separating hyperplane is found. In this paper we describe a simple, incremental LTU training procedure that, under reasonable, but not all training conditions, empirically performs near the theoretical optimum over all LTU learning procedures.

The paper has the following organization: First the standard LTU structure and the Basic LTU training algorithm are reviewed. Classes of linearly separable functions, their required weight size and training time are discussed. The Reflection algorithm, a variant of the Relaxation method of solving linear inequalities, is then described. The empirical behavior of the Basic and Reflection algorithms are compared on different types of functions and with different presentation orders. The issue of generalization accuracy and the possibility of combining multiple models to improve performance is considered next. As a possible explanation for the superior results of the Reflection algorithm, it is shown that eliminating over-generalization from the Basic algorithm produces results similar to Reflection. The time complexity of the Reflection algorithm is then investigated. The Winnow algorithm for adjusting LTU weights is discussed in light of its superior time complexity on irrelevant features and a combined Winnow/Reflect algorithm is proposed. In the final section, performance in terms of the number of different misclassified input patterns, rather than the total number of mistakes made is considered.

2 Basic Perceptron

2.1 LTU structure

An LTU is defined by a weight vector, $W = w_1, ...w_d, w_{d+1}$ of d real-valued feature weights plus a threshold weight. A feature vector, $F = f_1, ...f_d, 1$, to be classified by W, consists of d real-valued features plus a constant feature (generally equal to 1 or -1) corresponding to the threshold weight. Restricting the weights in W to integers does not affect its representational power assuming the set of feature vectors in finite. We will be primarily concerned with binary (two-valued) features, but the two-valued results can be generalized to n-valued features. Binary-valued features are traditionally either (0,1) or (-1,1).

A feature vector is classified as a positive or negative example of the category defined by W by taking its dot product with W and comparing with zero, i.e.

$$\begin{array}{ll} \text{if } F \cdot W > 0 & \text{then output 1} \\ & \text{else output 0 (or -1)} \end{array}$$

Notice that any positive linear combination of solutions is still a solution. Consequently the set of solutions is convex. Also this permits multiple models methods to create a solution which remains as a linear combination of the features. A weight vector which is not a solution vector but is the limit of solution vectors is called a *boundary* vector. The zero vector is such a point. A sequence of weight vectors may move towards a solution, but converge only to a boundary vector, yielding a non-solution.

2.2 Training

For training purposes we allow three output values (-1, 0, 1) corresponding to negative, zero and positive dot products. An output of 0 is always considered to be an error. That is, positive examples (class members) must give positive output and negative examples (non-members) must give negative output.

The Basic perceptron training algorithm (Rosenblatt, 1958; Nilsson, 1965; Minsky & Papert, 1969) can be used to find correct LTU weights given a finite set of positive and negative examples, also called input vectors or input patterns:

$$\begin{array}{l} \text{If } F \cdot W \text{ is too high, } W \leftarrow W - \alpha F \\ \text{If } F \cdot W \text{ is too low, } W \leftarrow W + \alpha F \end{array}$$

The constant α is an arbitrary, fixed, positive value, generally chosen to be 1 and so can be ignored. The initial weight vector is arbitrary, but is usually assumed to be the zero vector. If there exists a solution weight vector that separates the positive and negative input patterns, that is if they are linearly separable, the Basic algorithm is guaranteed to produce such a vector in a bounded number of corrections.

An upper bound on the number of misclassifications/adjustments the Basic algorithm requires to learn an LTU function is provided by a convergence proof (Nilsson, 1965). Given a solution weight vector, W, the function that it represents can be learned in at most $M|W|^2/a^2$ adjustments, where M is the squared length of the longest input vector and a is the smallest value of $|F \cdot W|$ over all F. $F \cdot W = 0$ is not possible or W would not be a solution vector, and assuming $\alpha = 1$ in the perceptron training rule and integer-valued input features, all weights will be integers and a will be at least 1. For (-1,1) input,

input vectors are all the same length and M is $d+1$. Thus an upper bound on learning is $(d+1)|W|^2$. The average length of the minimum integer solution weight vector for LTU functions of d features grows exponentially with d (see next section), so $|W|^2$ is the dominant term, and for simplicity, the effect of $(d+1)$ will be ignored. Consequently, to a reasonable approximation, the upper bound on training time grows as $|W|^2$.

Since the weights are adjusted in unit steps, a lower bound on learning time is simply the size of the largest weight in W where we assume the integer vector W has the smallest size possible. In fact it can be shown that learning time must grow at least as $|W|^2/(d+1)$ (Lewis II, 1966; Schmitt, 1996). Again this is dominated by the $|W|^2$ term. Since both the upper and lower bounds grow as approximately $|W|^2$, actual perceptron training time is likewise constrained. It is possible to extend this type of analysis to the case where only a portion of the input space is used for training (Volper & Hampson, 1987; Hampson, 1991) but that case is not considered here.

This result applies whether the input features are (0,1) or (-1,1). However, (-1,1) input generally leads to faster learning than (0,1) and it is used in all of the experiments reported here since the emphasis is more on actual application than on worst-case performance. If perceptron training time is used as a measure of unknown required weight size, (0,1) input is preferable since its empirical behavior more closely reflects the theoretical upper bound on training time, and thus the size of the required weights.

2.3 Bounds on LTU weight size

Since performance of the Basic algorithm, as measured by the number of adjustments made before a solution vector is produced, depends on the size of the weights required to represent the function, it is useful to consider the required weight size and the resulting training time for different classes of functions. Specifically, consider the functions in Table 1 and when represented as an LTU, their required weight size as a function of d. They provide points of comparison between theoretical bounds and empirical behavior and for empirical comparison of different learning algorithms. They also address the question of average performance. For simplicity of discussion, only approximate values are used here rather than the most precise bounds known.

The AND/OR function provides a convenient, "hard" LTU function to analyze and measure actual performance on. It can be written as:

$$(...(((F_1 \ and \ F_2) \ or \ F_3) \ and \ F_4)...) \ or \ F_d \quad (for \ odd \ d)$$

and is minimally represented as a weight vector whose weights are a Fibonacci series (Hampson & Volper, 1986). Since the ratio of succeeding terms in a Fibonacci series approaches the golden ratio ($\frac{1+\sqrt{5}}{2} \approx 1.618$), required weight size likewise grows exponentially

Table 1: Required weight size as a function of dimension d.

Function	Required LTU Weight Size
AND/OR	1.6^d
Muroga function	2^d
Average LTU function	$> 2^{d/2} \approx 1.4^d$
Any LTU function	$< 2^{d\ log(d)/2} = d^{d/2}$

with the number of features, i.e. at about 1.618^d. The size of the threshold depends on whether d is odd or even. Muroga (1971) describes a related function whose weights grow as a Fibonacci series but whose threshold grows smoothly with d. Muroga (1971) also provides a slightly more complex function construction method in which the weights grow as approximately 2^d. This is the most complex function we are aware of that can be constructed for any value of d. Hastad (1994), by a much more complex method, produces a function whose weights grow as approximately $d^{d/2}$ where d is a power of 2, although we have not implemented this as a test function.

The required weight size for the average LTU function of d features is not known precisely, but is exponential with d and is at least $2^{d/2} \approx 1.414^d$ (Hampson & Volper, 1986). The actual growth rate may depend on d since the opportunity for increasingly hard LTU functions increases with d. For example, the AND/OR function is (obviously) no harder than AND or OR until $d > 2$, and Muroga's function is not harder than the AND/OR function until $d > 8$. So, unlike the AND/OR function which has a fixed weight complexity of approximately 1.6^d, the complexity of the average LTU function may asymptotically approach its maximum value as d increases. An upper bound of approximately $2^{d\ log(d)/2} = d^{d/2}$ on required weight size for any LTU function has been shown (Muroga, 1971).

Based on the $|W|^2$ growth rate, for large d the expected training time for the AND/OR function grows as approximately $(1.618^d)^2 \approx 2.618^d$. Training time for random functions grows faster than $(2^{d/2})^2 = 2^d$ (using the lower weight bound for random functions) and may not be greater than $(2^d)^2 = 4^d$ (using Muroga's function as a plausible upper bound on average complexity). Using Muroga's upper bound on LTU weight size produces an upper bound on training time of approximately $2^{d\ log(d)} = d^d$. The general conclusion is that if perfect classification of the complete input space is required, Basic Perceptron training time is exponential in the number of features.

3 The Reflect algorithm

We now describe a slightly different algorithm for learning LTU weights. As in the Basic perceptron training algorithm, Reflect adds a multiple of the misclassified example F. However the multiple α is not a constant 1 as in Basic , but depends on $F \cdot W$. Intuitively Reflect adds in just enough of F to "reflect" the output (same magnitude, sign changed). More precisely the update rule is defined below.

$$\begin{array}{ll} \text{If } F \cdot W \text{ is too high,} & W \leftarrow W - \alpha F \\ \text{If } F \cdot W \text{ is too low,} & W \leftarrow W + \alpha F \\ & \text{where } \alpha = \lambda \frac{|F \cdot W|}{F \cdot F} \text{ and } \lambda = 2 \end{array}$$

This is known as a "Relaxation" method of solving linear inequalities (Agmon, 1954; Motzkin & Schoenberg, 1954; Nilsson, 1965; Duda & Hart, 1973), and can be shown to converge if the amount of correction, λ, is between 0 and 2 ($0 < \lambda <= 2$). Clearly $\lambda = 0$ produces no change in output, $\lambda = 1$ corrects output to exactly 0, and as described above, $\lambda = 2$ "reflects" the output. With $\lambda = 0$ or $\lambda = 2$, the length of the weight vector remains unchanged by an adjustment, while with $0 < \lambda < 2$ it shrinks with each adjustment. In fact, if one demands that the value of α be such that the resulting W is correct on F and is also the length as the original W, then λ will be 2. When $\lambda = 1$ the maximum amount of shrinkage is produced. For $\lambda = 2$ the algorithm is guaranteed to terminate after a finite number of steps (Nilsson, 1965 p 93), while for $\lambda < 2$ it may simply converge on the boundary of the solution space (Motzkin & Schoenberg, 1954 p 396). The specific case of $\lambda = 2$ has been called the "Reflexion" method (Motzkin & Schoenberg, 1954). We will refer to our implementation of this adjustment rule, plus some necessary modifications, as the Reflection algorithm, or for brevity, Reflect.

There are two problems in applying the algorithm as stated. First, there is the special case of what to do when $F \cdot W$ is exactly 0 and thus cannot be reflected. This occurs trivially when the initial vector is the zero vector, but can occur at other points in the training process. The second problem is less obvious: while the sequence of adjustments to the weight vector is guaranteed to terminate in bounded time, there is no guarantee about the numerical precision that is required to actually implement the algorithm. This can be a problem because the amount of correction on each adjustment (the amount of F that needs to be added or subtracted) tends to shrink with the number of adjustments, and can become so small that it falls within the roundoff error of fixed-precision variables. In this case the algorithm may stall. We call this the *shrinkage/roundoff problem.* Such roundoff error can occur at any point in the training sequence but increases in probability as the size of the adjustment shrinks. The actual size of the adjustments and the rate of shrinkage is highly variable and depends on a number of factors. The effect of the adjustment shrinkage is that the candidate weight vectors converge toward the boundary of the solution space. The possibility of serious roundoff error is, of course, not unique to the Reflection algorithm. The Basic algorithm can also encounter roundoff problems in the form of integer overflow, but in practice it is never a problem.

We have developed two versions of the Reflection algorithm which handle the shrinking adjustment problem in different ways. In both versions a Basic adjustment (ie $\alpha = 1$) is done when $F \cdot W = 0$ or when adjustments appear to be getting too small for accurate correction. As a related issue, a lower bound on adjustment size prevents the algorithm from converging on the boundary of the solution space. The hope is that by interleaving two convergent approaches the result will still be convergent. The key point in obtaining fast convergence is that the non-relaxation steps are only rarely applied. The main difference between Reflect 1 and 2 is that while both must deal with the case of actual or imminent roundoff error, Reflect 1 tries to control weight size in such a fashion that this seldom occurs.

3.1 Reflect 1

In Reflect 1, the algorithm is modified to try to keep the adjustments from shrinking to the point that roundoff error actually becomes a significant problem. Three changes were made to achieve this:

1) Use $\lambda = 2.001$ instead of 2.0

That is, reflect to a slightly larger magnitude than the original output. The value of λ is one of several factors that affect the rate of shrinkage, with the shrinkage rate increasing as λ decreases from 2.0 to 1.0. Values of λ greater than 2.0 can lead to growth in adjustment size. Perhaps not surprisingly, while $\lambda < 2.0$ causes the length of the weight vector to shrink on each adjustment, $\lambda > 2.0$ causes it to grow. A value slightly above 2.0 may only slow the rate of adjustment shrinkage, but this helps during extended training sequences. It has essentially no impact with normal input ordering, but helps with LeastCorrect input order (see below). In addition, there is occasionally some value in using a non-integer value for λ since the resulting weights are less apt to produce outputs that are exactly zero. In most cases, $\lambda < 2.0$ leads to slightly faster convergence, but with faster adjustment shrinkage and a correspondingly greater risk of encountering serious roundoff error. Using a somewhat larger value of λ thus trades a slight reduction in normal performance for a reduction in the more rare occasions where roundoff error becomes a significant problem.

Simply using $\lambda = 2.0$ is adequate in almost all cases, but shrinking adjustments is a problem that needs to be dealt with one way or another and adjusting λ a bit is an easy (if inelegant and only partially effective) method of addressing it. As previously observed, the convergence proof is only for the region $0 < \lambda <= 2$, but in conjunction with the third modification, λ can be increased beyond that without dire consequences.

2) If $\alpha < 10^{-10}$, then $\alpha = 1$

A second modification is to simply do a Basic adjustment whenever the value of α becomes too small. Specifically, this is to avoid small adjustments that might cause roundoff problems, despite the first modification. An adjustable lower bound on the magnitude of α only indirectly controls the required precision of the weights, but is quite effective in providing an adjustable precision parameter. Adjusting this parameter permits exploration of precision/performance tradeoffs. This also takes care of the special case of $F \cdot W = 0$ since α is zero then. The specific cutoff value of 10^{-10} was chosen somewhat arbitrarily and requires double precision variables.

Frequent application of this rule can, and generally does, slow down learning, so while values larger than 10^{-10} are acceptable, learning may be slower. As the lower bound on α approaches 1, the algorithm behaves more like Basic perceptron training.

3) If $\alpha > 1$, then $\alpha = 1$

The final modification is to keep values of $\lambda > 2.0$ from causing an explosive growth in weights. What it means is to never add in more than a Basic adjustment. Again, this has little impact on performance when λ is near 2.0, but it allows λ to be increased beyond 2 without fear of an explosion in weight size. For large values of λ the algorithm acts more like Basic perceptron training.

3.2 Reflect 2

As an alternative to avoiding roundoff problems by trying to control the rate of adjustment shrinking, the adjustment size can be allowed to shrink unimpeded to the point that roundoff error actually becomes significant, and only then address the problem. Specifically, a Basic adjustment is done when serious roundoff error is detected. For example, roundoff error can be detected when an adjustment fails to change the sign of the output. Unfortunately there are circumstances where the algorithm stalls due to roundoff error even though it successfully corrects the sign of the output on each adjustment. A slightly stronger test is to see if the adjusted output is within 1% of the target "reflected" value. There may be circumstances that slip by this test too, but in practice it works quite well. Like adjusting α in Reflect 1, this also provides an opportunity to adjust the required precision by adjusting the minimum accuracy of the adjustment. This rule, with double precision variables, $\lambda = 2.0$ and a special test for $F \cdot W = 0$ generally produces slightly superior results than Reflect 1 because, in most cases, roundoff error never becomes a problem. It also has the advantage of automatically adjusting to the precision of the variables. Consequently, for simple applications, Reflect 2 is the preferred version. However, because it treats λ and the minimum magnitude of α as adjustable parameters, Reflect 1 has been useful for experimental investigation. The empirical results of Reflect 1 and 2 are quite similar. The results presented in this paper are for Reflect 1.

3.3 Input ordering

The upper and lower bounds on Basic adjustments are independent of the presentation order of input patterns, but presentation order can have a sizable impact on empirical results (Hampson & Volper, 1986; Hampson, 1991) and this is especially true for the Reflect algorithm. Because of this dependence, it useful to try a range of presentation orders in order to characterize the performance of a training algorithm. Specifically, consider the following ordering strategies:

ShuffleCycle: randomize input order after each cycle through the input space
FixedOrder: pick a random order and continually cycle through it
Numeric: order the binary input patterns as though they were binary numbers
LeastCorrect: adjust on mistake closest to the solution hyperplane
MostCorrect: adjust on mistake farthest from the solution hyperplane
LongAdj: adjust on mistake farthest from current, learned hyperplane

For LeastCorrect and MostCorrect, ties are broken systematically (see below) while for LongAdj they are broken randomly.

ShuffleCycle and FixedOrder give similar results, so either can be used as the "standard" ordering technique. However, ShuffleCycle seems the least biased of the two and to the extent that they differ, it is generally faster, so it is used in the experiments reported here. As an aside we note that, in general, one should avoid introducing any unncessary bias into an algorithm. Conversely, removing unjustified biases appears to improve the performance of numerous algorithms and, in particular, those that do heuristic search (Hampson & Kibler, 1996). Numeric order is a specific case of fixed ordering and for the Basic algorithm gives similar results, but for the Reflect algorithm it is generally much slower than a fixed, but initially randomized presentation order.

Least and MostCorrect are only possible if you already know a solution vector, so they are not practical training strategies, but they are useful for testing the extremes of algorithm performance. They are generated by sorting all input patterns by their distance to the known solution hyperplane. Presentation starts at the beginning of the list and continues until a mistake is made and adjusted. Presentation then restarts at the beginning of the list. For LeastCorrect order they are sorted in increasing order, while with MostCorrect they are sorted in decreasing order. Because it minimizes the average "a" in the upper bound, $M|W|^2/a^2$, LeastCorrect produces very slow learning and generally reflects the upper bound on training time, which assumed $a = 1$. Conversely, MostCorrect maximizes a on each adjustment and so would be expected to lead to faster than average convergence. MostCorrect is in fact somewhat faster than ShuffleCycle for Basic training, but because it is typically adjusting on patterns that are almost correct in output already, it sometimes produces a series of errors with zero output. Under these conditions, Reflect acts more like the Basic algorithm.

LongAdj order generally produces faster than average learning since a few big adjustments can often do the work of a larger number of small adjustments. The initial proof of convergence of the Relaxation method (Motzkin & Schoenberg, 1954) was for LongAdj order although it was subsequently shown to converge for any ordering (Nilsson, 1965). With binary features, LongAdj often produces the best performance from the Reflection algorithm. This is useful when comparing "best case" performance with more practical presentation orders such as Shuffle or Fixed order.

Many other orderings strategies are possible, and the results are often interesting, but the extremes of algorithm performance are reasonably well covered by the limited selection considered here. For example, in Figure 1, the performance of the Basic algorithm is compared with MostCorrect, ShuffleCycle and LeastCorrect order. In this experiment the average percent accuracy over the entire training set of 2^d patterns is shown vs the number of adjustments for 100 random LTU functions of 10 features. (the generation of random LTU functions is discussed in the following section) As evident in the graph, MostCorrect and LeastCorrect produce very different learning curves and ShuffleCycle is similar to MostCorrect.

4 Empirical Results

As previously observed, the weights of the AND/OR functions grow as 1.618^d, so learning time for Basic training, as measured by the number of adjustments, grows as about $(1.618^d)^2 \approx 2.6^d$. In order to test this rough prediction, learning time was tested with increasing d. For ShuffleCycle order, the AND/OR function was learned 100 times for each value of d. For LeastCorrect order it was learned once. All 2^d patterns were used as training instances and training continued until all were classified correctly. Tests were run with d varying between 2 and 16. The observed growth rate was more consistent with larger d, so the average growth rate in adjusts was calculated for d between 8 and 16. The results are shown in Table 2.

As seen in Table 2, the empirical results for Basic training time using both ShuffleCycle and LeastCorrect order are in agreement with the theoretical prediction of approximately 2.6^d. The more interesting result is that the growth rate for the Reflection algorithm with ShuffleCycle order is not only well below the predicted rate for the Basic algorithm, but below the 1.6^d growth rate of the required weights. The actual growth rate of the Reflection algorithm under these conditions will be investigated in greater detail in section 6, where it is shown that a polynomial rather than exponential growth rate provides a better

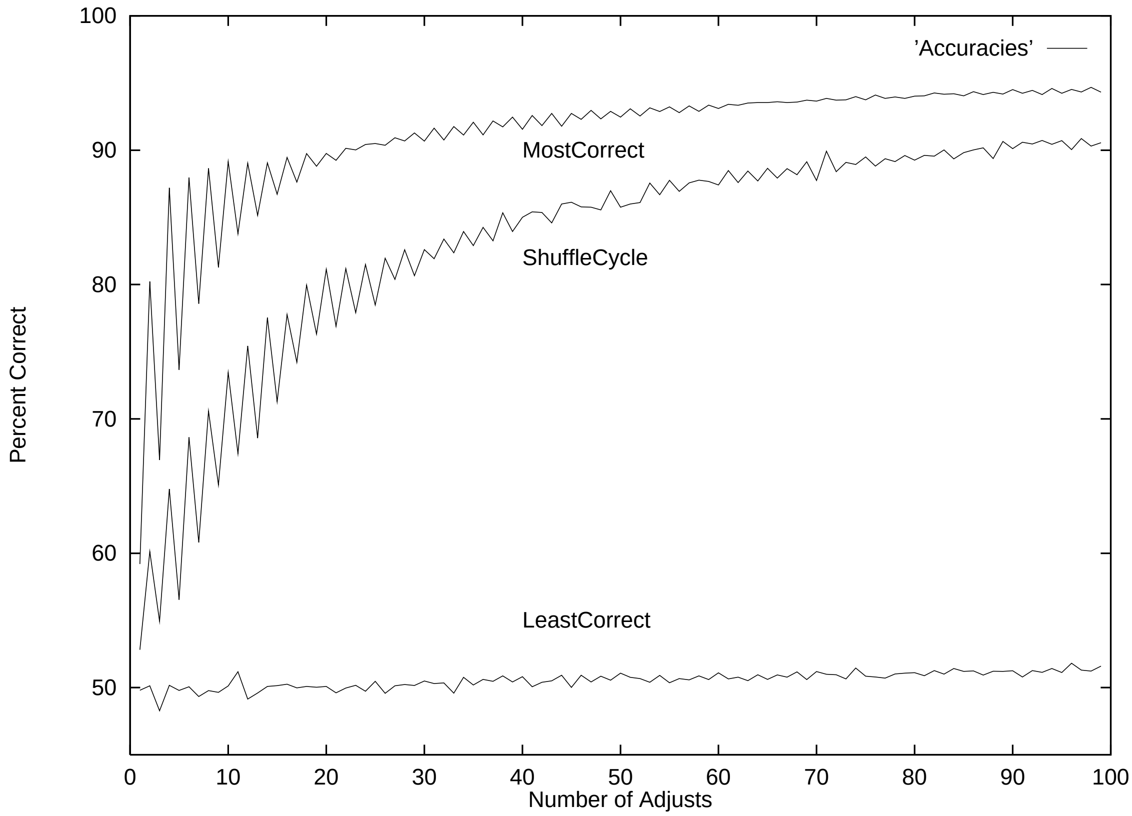

Figure 1: Average classification accuracy vs number of adjustments for the Basic algorithm using MostCorrect, ShuffleCycle and LeastCorrect orders. 100 random functions, 10 features. Adapted from (Hampson 1991)

Table 2: Growth rates in adjustments for Basic and Reflect on AND/OR function.

Ordering	Growth Rate
Basic ShuffleCycle	2.47
Basic LeastCorrect	2.84
Reflect ShuffleCycle	1.20
Reflect LeastCorrect	2.34

Table 3: Growth rates in adjustments for Basic and Reflect on random LTU functions.

Ordering	Growth Rate
Basic ShuffleCycle	3.26
Basic LeastCorrect	3.63
Reflect ShuffleCycle	1.26
Reflect LeastCorrect	2.56

fit. With LeastCorrect order, Reflect performs more like the Basic algorithm, but is still significantly faster. It should be noted that the poor performance of the Reflect algorithm using LeastCorrect order is not due to roundoff errors or the modifications designed to avoid roundoff; the unmodified algorithm performs equally poorly even in those cases where roundoff problems do not occur.

The AND/OR function is useful as one point of comparison but performance on a wider range of tests is also of interest. Although not shown here, Muroga's function gave comparable results except that the Basic growth rate was about $(2^d)^2 = 4^d$ rather than 2.6^d.

In order to roughly gauge what the performance of the "average" LTU function would be, random LTU functions were created by generating a solution weight vector with random integer weights between -10^7 and 10^7. This solution vector was then used to classify the input patterns and to order them for LeastCorrect training order. For each value of d (2 .. 15), 100 functions were generated this way. (Note that choosing an LTU weight vector with weights chosen uniformly at random is not necessarily the same as choosing an LTU function uniformly at random from the set of all LTU functions, although empirical results suggest that it is a good approximation.) Each function was learned once with ShuffleCycle and once with LeastCorrect ordering. The required weight size for average LTU functions is not known exactly, but Basic training time is at least $(2^{d/2})^2 = 2^d$, and may approach $(2^d)^2 = 4^d$. It cannot be worse than $2^{d\ log(d)} = d^d$.

The growth rates over the range $d = (7 \ldots 15)$ are show in Table 3. For Basic, the results are well in excess of the lower bound of 2^d. On average, for $d > 8$, randomly generated functions are empirically harder to learn than the AND/OR function, so for large d, their required weight size is certainly bigger than the lower bound of 1.4^d and probably bigger than the 1.6^d growth rate of the AND/OR function. As d increases, it may well approach 2^d. As with the AND/OR function and Muroga's function, Reflect with ShuffleCycle order does better than the lower bound of Basic training although performance degrades significantly with LeastCorrect order.

The raw results in terms of the average number of adjusts and cycles needed to learn random LTU functions are shown in Table 4. It is apparent that the slower growth rate

Table 4: Average adjustments and cycles for Basic and Reflect. 100 random functions, ShuffleCycle order, increasing d.

Features	Basic		Reflect	
	Adjusts	Cycles	Adjusts	Cycles
2	2.26	1.0	2.26	1.00
3	7.19	2.12	7.19	2.12
4	11.22	2.89	6.84	1.83
5	32.65	7.35	11.07	2.18
6	69.73	14.19	17.55	2.61
7	233.07	32.52	25.35	2.78
8	569.54	70.93	34.71	3.07
9	2101.60	219.47	46.30	3.61
10	5802.02	548.52	60.59	3.46
11	22518.61	1652.32	76.02	3.61
12	66655.61	4486.78	95.99	3.80
13	262282.82	15557.13	116.38	3.98
14	783346.56	42750.50	138.68	4.15
15	2978862.83	136118.01	163.20	4.23

of Reflect rapidly translates into a sizeable difference in absolute numbers. Besides the reduction in adjusts, the reduction in cycles is equally dramatic. This value more closely reflects the reduction in actual runtime.

4.1 Classification Accuracy and Model Averaging

The number of misclassifications/adjustments needed to train an LTU to correctly classify a set of patterns is a standard measure of algorithm performance, but it is also interesting to ask how the weight vectors learned by Basic and Reflect compare in accuracy of generalization. This can be tested in various way, but perhaps the most direct method is to train on a randomly chosen subset of the input patterns, and then test the resulting weight vector on the remaining, unseen patterns. Because both algorithms learn a separating hyperplane for the same training set, we expected that both would produce equal accuracy of classification on the test set. It was somewhat surprising then when Basic showed a slight, but consistent advantage in testing accuracy. This was true over different functions, varying number of features and varying training/testing partition size, although the magnitude of the effect varied with the testing conditions. More specifically, generalization accuracy (percent correct classification over the test set) increased for both algorithms with the number of features and the training/testing partition ratio. Holding either factor constant and increasing the other increased generalization accuracy, so the absolute difference between algorithms decreases as those factors increase. For random functions using ShuffleCycle ordering and a wide range of d and training/testing partitions, Basic was seldom, if ever, more than 3% more accurate and most often less than 1%, but it was consistently better when averaged over a large number of tests.

One general technique for improving classification accuracy is to combine multiple models. At a minimum this reduces the variance inherent in the set of possible models for a given training set, but usually improves generalization accuracy over the average accuracy of the models as well. This approach is relatively easy to apply to LTU learning since different input orders of the training set lead to different solution weight vectors and these solution vectors can simply be added or averaged to produce a new, combined model. Applying this approach to the train-and-test paradigm does in fact improve generalization accuracy for both Reflect and Basic, with the two algorithms now giving comparable results. Improved accuracy could generally be achieved by combining as few as 10 models, although 100 gave more consistent results. Again the improvement in accuracy varied with a number of factors but was generally no more than a few percent.

Performance on non-linearly separable functions is also of interest in this context. In this case one measure of accuracy is what a weight vector's non-zero error rate is over the entire training set. This was tested by taking a linear function such as AND/OR and reversing the classification of a certain number of randomly chosen input patterns. The minimum error rate for an LTU over this training set is no more than the number of reversed patterns, and might be slightly less. In practice however, the error rates for the infinite series of weight vectors learned by both Reflect and Basic is well in excess of this, with Reflect being significantly worse. The accuracy advantage of Basic in this case was not entirely unexpected. Reflect makes large adjustments on patterns that produce large errors while Basic always makes fixed-sized adjustments. The result is that with Reflect, the hyperplane shows a much larger "wobble" in response to the non-linear points, which degrades average performance over the large majority of separable points.

The "perceptron cycling theorem" (Minsky & Papert 1969 p. 181) states that although training does not terminate in the non-linear case, the length of the weight vector remains bounded. Consequently, when the Basic algorithm is used, the weight vector is limited to cycling among a finite set of integer-valued vectors, and the "wobble" of the hyperplane does not decrease with extended training. Using Reflect and $\lambda = 2$, the weight vector is not only bounded, but constant in length, although it is not limited to integer values. More importantly, because Reflect always moves the hyperplane enough to correct each misclassified point, its wobble cannot decrease with extended training either. This undamped wobble means that neither training algorithm can converge on a minimum-error separator.

This problem can be addressed by the same model-averaging method used to reduce variance and improve accuracy in the previous train-and-test example. However, in this case, multiple training episodes are not required to produce the multiple models; since convergence is never achieved, continued training produces a series of weight vectors which can simply be summed. While the actual hyperplane continues to wobble during training, the sum of the weight vectors results in a hyperplane that does stabilize with an accuracy that is often better than any of the individual weight vectors. Reflect and Basic are both greatly improved by this method although Basic still retains an advantage. The resulting error rate approaches but does not achieve the minimum error rate.

It is interesting to note that, for Basic, this approach of summing the series of weight vectors also leads to a faster improvement in accuracy in the linearly separable case, although the sum of the series is not guaranteed to produce perfect classification by the time the final weight vector in the series does. For Reflect, the individual weight vectors are more accurate than the sum of the series in the separable case.

Similar results are obtained in the presence of random noise, either in the input features or in the classification training signal. Because of the noise, training continues indefinitely as in the non-linear case, and the average accuracy of the individual weight vectors in the series is better for Basic than for Reflect. Again, model averaging (summing the series of weight vectors) works well for both Reflect and Basic, and the error of the long-term sum approaches zero.

There is an inherent correlation between learning speed and increased sensitivity to "bad" input patterns, and the sensitivity of Reflect to noise or non-linear points may prove to be its most significant limitation in application to real-world domains. Model-averaging helps, but does not completely eliminate the problem. It is not our intent to explore this topic in detail here, but it is an important issue that deserves further consideration.

5 Why Reflect Works Well

Given the impressive convergence speed advantage of the Reflect algorithm, it is interesting to ask what fundamental property distinguishes it from the closely-related Basic algorithm, and whether a general principle can be extracted.

We have investigated the possibility that the Reflect algorithm better approximates a "rule of minimum generalization" than the Basic algorithm. That is, given a generalization strategy, (adding F to W in this case) never generalize more than is minimally necessary to correct a misclassification. It is apparent that just adding in the input vector, as Basic does, can either under- or over-correct: it may fail to add in enough to correct the output on the current input, or it may add in so much that it changes the classification of more input patterns than minimally necessary. On the surface, both of these seem to be suboptimal strategies. We will use the terms over-correction and over-generalization (OG) interchangeably. In absolute numbers, the Reflection algorithm makes fewer over- and under-corrections than Basic, but it also makes far fewer adjustments, so it is not obvious if this is a cause or an effect. In order to investigate this point, we tried to minimally modify the Basic algorithm to remove under- and over-correction.

Under-correction is easy to detect and fix. For example, make all corrections so that the new, corrected output is exactly +1 or -1, or any fixed value $+b$ and $-b$ (Duda & Hart 1973 p 148). This guarantees that there are no under-corrections. It is generally faster than the Basic algorithm, but not dramatically so. Empirically, under-correction seldom occurs anyway, and it does not appear to be a major deficiency of the Basic algorithm, at least under the testing conditions used here.

Over-correction is more expensive to detect and avoid. However, it does occur frequently (with ShuffleCycle ordering about half the time) and appears to be the primary issue. To test this, we wrote the MinAdj algorithm which is designed to avoid both under- and over-correction. Specifically, it first adds enough to move the hyperplane so that it goes through the currently misclassified point. It then adds in enough to move it a fixed fraction P $(0 < P < 1)$ of the distance that would cause the next point to change classification. $P > 0$ guarantees that the misclassified point is corrected and $P < 1$ guarantees that no points unnecessarily change classification. Note that the MinAdj algorithm with $P = 0$ is equivalent to the Reflect algorithm with $\lambda = 1$; they both move the hyperplane to pass directly through the misclassified point. Unfortunately the MinAdj strategy is rather expensive to implement, requiring a scan through all the input patterns for every adjustment.

Empirically, MinAdj results are in fact quite similar to those of the Reflect algorithm,

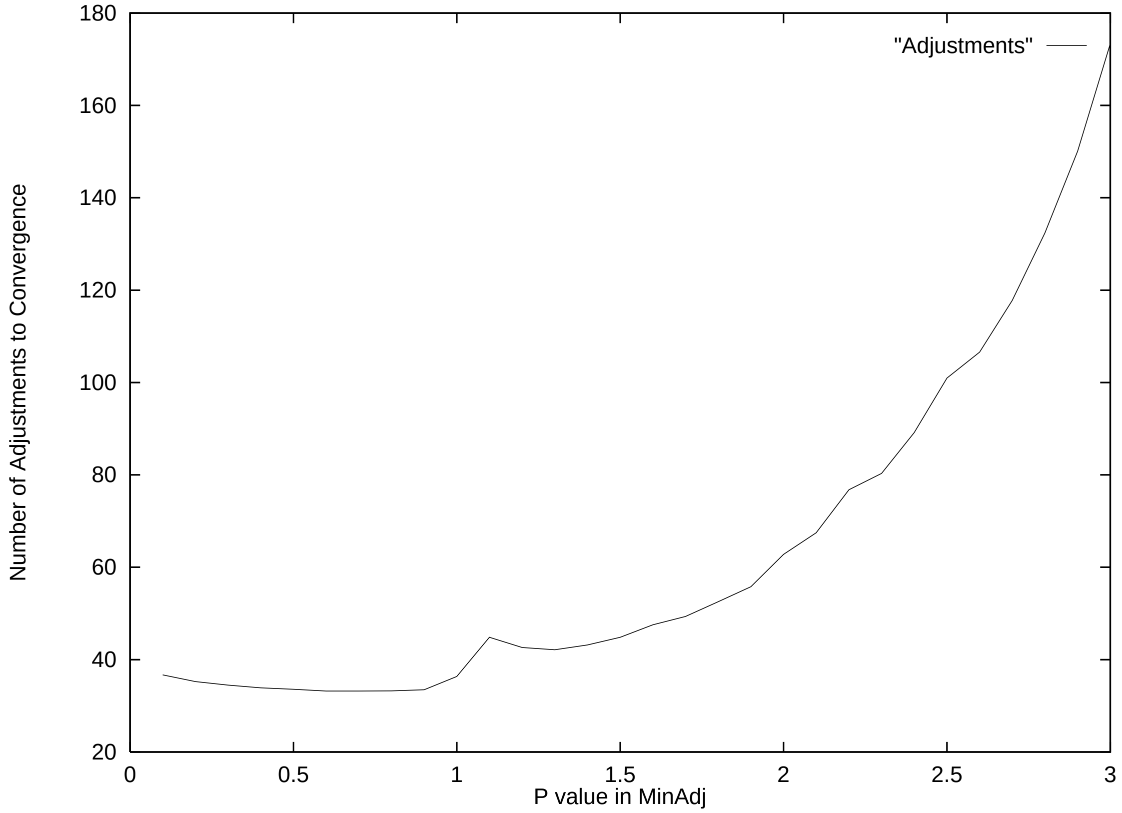

Figure 2: Number of adjustments to convergence versus degree of over-generalization, averaged over 100 random functions with 8 features using MinAdj, ShuffleCycle order, P in 0.1 increments.

supporting the view that the power of the Reflect algorithm comes from a reduction in OG. Further support is provided by the fact that the MinAdj algorithm is relatively insensitive to the value of P in the range $0 < P < 1$, but that performance, as measured by the number of adjustments to convergences, degrades significantly between .99 and 1.01 where OG first occurs. What this means is that while any value of $P > 0$ is an over-correction in the sense that more than a minimum amount of the input space changes classification, the amount of over-correction doesn't noticeably affect performance unless observed input patterns are in that region of the input space. In addition, as seen in Figure 2, the algorithm still converges, but performance continues to degrade as P is increased beyond 1. In this experiment, 100 8-feature random functions were learned with the MinAdj algorithm using ShuffleCycle ordering and a range of P values in .1 increments.

Based on these results, it appears that over-generalization is a significant deficiency of the Basic algorithm, since it both occurs frequently and is detrimental to performance. The Reflection algorithm might therefore be viewed as an inexpensive approximation of the ideal MinAdjust strategy. More likely however, they are both approximations of some even more optimized adjustment strategy for incrementally adjusting an LTU weight vector. For example, the fact that MinAdj is not uniformly superior to Reflect suggests that there are circumstances where OG is useful. We have not explored this point, but have observed MinAdj training situations where adjustment on two nearly identical patterns is locked in tight alternation, producing very slow learning. An occasional over-generalization might help to keep such pathological adjustment sequences from becoming established. In general, the MinAdj algorithm is much more sensitive to ordering effects than the Reflect algorithm.

Besides the obvious impracticality of always determining the minimum adjustment, the MinAdj algorithm suffers from the same shrinking-adjustment/roundoff problem as the Reflect algorithm. For ShuffleCycle order, this is not much of a problem, but like the Reflect algorithm, roundoff problems soon set in when using LeastCorrect ordering. Moving the hyperplane more than half way to the next point (e.g. $P = .95$) helps somewhat but does not solve the problem. As in the Reflect 1 algorithm, this was avoided by setting a lower bound of 10^{-10} on the size of α, although the "1%" rule of Reflect 2 would be equally applicable.

Despite the considerable circumstantial evidence that OG is a major rate-limiting factor for the Basic algorithm, and that its elimination produces behavior much like the Reflection algorithm, it remains a conjecture that the superior performance of Reflection over Basic can be understood as resulting from a reduction in OG. In fact, by the most obvious measures of comparison (frequency or average amount of OG), Reflection shows no reduction. However, the average amount of OG decreases with the number of adjustments for both Reflect and Basic, and Reflect decreases faster (Figure 3). The reason for the decline is similar in the two algorithms. In the case of Basic, the weight vector increases in length as training proceeds, so the fixed-length adjustments become proportionally smaller. As the relative size of the adjustment to the size of the weight vector decreases, so does

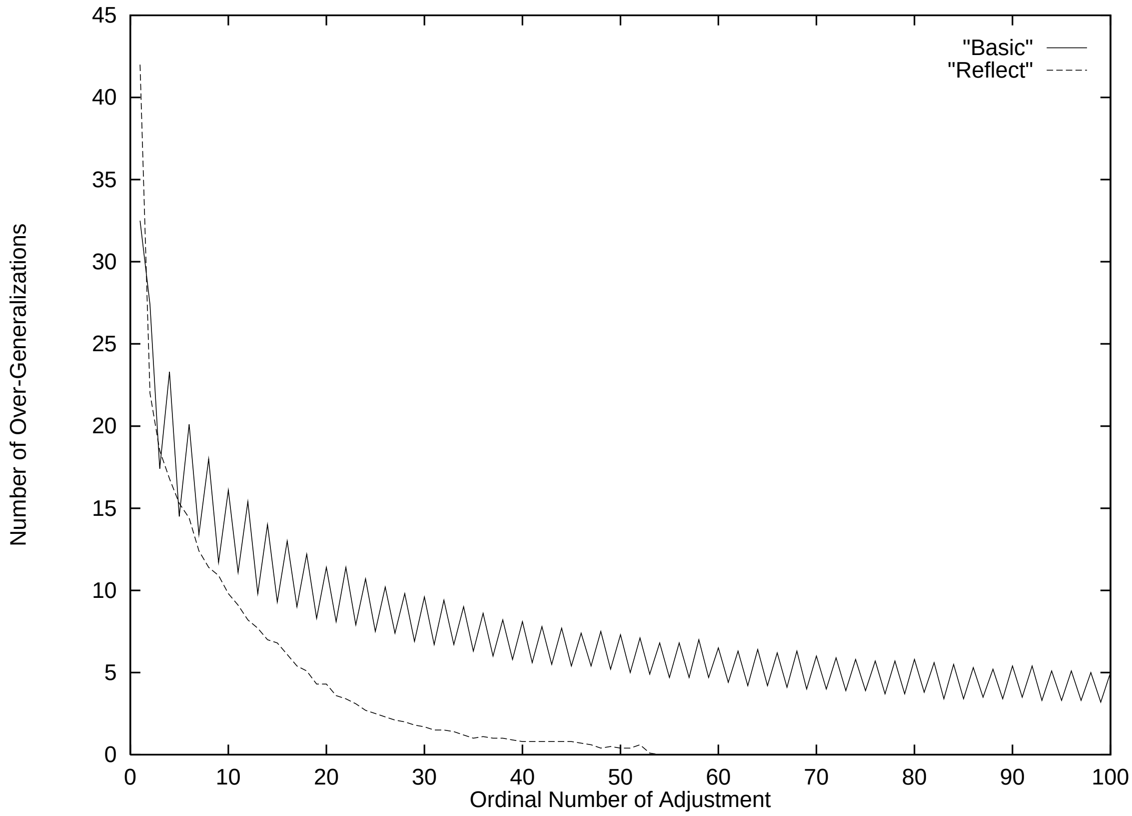

Figure 3: Average amount of over-generalization versus adjustment number for Basic and Reflect, averaged over 1000 random functions with 8 features, ShuffleCycle order.

the amount of OG. For the Reflection algorithm, the length of the weight vector remains unchanged during training but the size of the adjustment decreases. Again, this leads to decreasing OG. The advantage of Reflection appears to be that the ratio of adjustment to weight vector, and thus OG, drops more rapidly. This is the flip side of the shrinking adjustment/roundoff problem. In addition, with $\lambda > 1.0$, under-correction cannot occur with Reflect as is possible with Basic. This experiment was done by measuring the amount of OG as the number of patterns that unnecessarily changed classification on each adjustment. The results were averaged over 1000 8-feature random functions using ShuffleCycle ordering. When viewed this way it is not surprising that the average frequency or amount of OG per adjustment is not less for Reflect. In fact, because of its long tail, the average OG over the entire training sequence is smaller for Basic.

The Reflection algorithm has been the most successful to date, but if the goal is to avoid OG, without the computational overhead of the MinAdj algorithm, a number of other heuristics are at least partially effective.

For example, as seen in Figure 3, OG declines as training proceeds. As previously observed, this is because the adjustment becomes progressively smaller in relationship to the length of the weight vector. This effect can be simulated to some extent by starting the weights at large random values (e.g. $\leq 10^6$) rather than zero, and using exact correction to -1 and +1. Because of the large weights, the input pattern's outputs are initially widely spread out, and correcting to -1 or +1 is in fact usually a minimal correction. This works quite well for ShuffleCycle, but quickly succumbs to LeastCorrect. A fundamental problem is that the initial wide spread of the pattern's outputs slowly wears out, so that after many adjusts, things are more or less the same as if it had been started without big weights, and performance degrades toward Basic behavior.

As an alternative, use the Basic algorithm, but any time there is an input with an output less than 10 (or some function of d), multiply all weights by 1.01. Again the goal is that all outputs are sufficiently large that most adjustments will not cause OG. This works reasonably well some of the time, but runs the risk of explosive weight growth.

6 Mistake Bound of Reflect

The Reflection algorithm is quite similar to the Basic perceptron training algorithm, but clearly does not have the same time complexity since it is consistently and dramatically below the Basic lower bound established in section 2.

A different line of reasoning provides a different perspective which is based on optimal learning time over all possible LTU learning algorithms. In the simplest analysis, there are at least $2^{d^2/2}$ and at most 2^{d^2} LTU functions for d binary features (Muroga, 1971), so, using the lower bound, any learning algorithm must see (misclassify/adjust on) at least $log_2\left(2^{d^2/2}\right) = d^2/2$ points to distinguish all LTU functions. That is, the number of mistakes must be at least quadratic with d. A more precise analysis (Maass, 1994; Maass & Turan, 1994) shows that the lower and upper bounds on misclassifications for any optimum learning algorithm for LTUs are proportional to d^2 and $d^2 log(d)$. They describe a feasible algorithm that achieves the upper bound using linear programming techniques. The required precision of the algorithm's variables increases polynomially with d.

The lower bound on adjustments can also be easily, if impractically, achieved by the following construction: Maintain a list of all LTU functions that are consistent with the points seen so far and classify previously unseen points by majority rule. This guarantees that every misclassified point will eliminate at least half of the remaining hypotheses. Littlestone (1988) refers to this as the "Halving" algorithm. Since there are at most 2^{d^2} LTU functions, the Halving algorithm can make at most $log_2 2^{d^2} = d^2$ mistakes. The linear programming method described in Maass and Turan (1994) has a similar goal, but treats the LTU hypothesis space geometrically, and so does not require an actual enumeration of the individual hypotheses. In addition, their analysis requires that the learning algorithm always propose a valid LTU hypothesis (as do the Basic and Reflection algorithms); the

Table 5: Growth rate in adjustments for Reflect. 100 random functions, ShuffleCycle order, increasing d. ave = average number of adjustments, $ratio = ave_d/ave_{d-1}$, $d^2 = ave/d^2$, likewise for next two columns

d	ave	$ratio$	d^2	$d^2log(d)$	d^3
2:	2.26	–	0.565	0.815	0.282
3:	7.19	3.181	0.799	0.727	0.266
4:	6.84	0.951	0.428	0.308	0.107
5:	11.07	1.618	0.443	0.275	0.089
6:	17.55	1.585	0.487	0.272	0.081
7:	25.35	1.444	0.517	0.266	0.074
8:	34.71	1.369	0.542	0.261	0.068
9:	46.30	1.334	0.572	0.260	0.064
10:	60.59	1.309	0.606	0.263	0.061
11:	76.02	1.255	0.628	0.262	0.057
12:	95.99	1.263	0.667	0.268	0.056
13:	116.38	1.212	0.689	0.268	0.053
14:	138.68	1.192	0.708	0.268	0.051
15:	163.20	1.177	0.725	0.268	0.048
16:	189.71	1.162	0.741	0.267	0.046
17:	217.69	1.147	0.753	0.266	0.044
18:	247.54	1.137	0.764	0.264	0.042
19:	277.90	1.123	0.770	0.261	0.041
20:	314.20	1.131	0.786	0.262	0.039

voting strategy of the Halving algorithm does not conform to this restriction.

Although there is no a prior reason to believe that Reflect is in any sense optimal, this type of analysis at least establishes that LTU learning can be polynomial rather than exponential with d. In fact, for Reflect with random functions and ShuffleCycle ordering, the ratios of succeeding terms in the series of average adjusts consistently declines as d increases, suggesting that the series is less than exponential with d (Table 5, "ratio" column). The optimal-algorithm analysis provides lower and upper bounds of d^2 and $d^2log(d)$, so those were tested as well as d^3. In the last 3 columns of the table, the average number of adjusts is divided by d^2, $d^2log(d)$, and d^3 respectively. As seen in the table, the series grows faster than d^2, slower than d^3 and is approximated by $d^2log(d)$ quite well. In the same experiment, the average number of cycles to convergence appears to grow as approximately $log(d)$.

Measured this way, growth rates of the AND/OR function and Muroga's function are below and above $d^2log(d)$, respectively, although in absolute terms the results are all quite similar. The fact that Muroga's function results in a growth rate greater than $d^2log(d)$ emphasizes the fact that Reflect only approximates optimum learning and does not guarantee it, as with the linear programming approach. However, performance can be improved by using LongAdj ordering and reducing λ to about 1.6, and under these optimized conditions $d^2log(d)$ performance is achieved for Muroga's function as well.

With LeastCorrect order, the number of adjustments is much greater than with ShuffleCycle or FixedOrder, and Reflect appears to exhibit exponential growth. Baum (1990) reports similar results in the PAC learning context, where malicious input selection produces exponential learning time and nonmalicious selection leads to polynomial time (see section 9.0). To some extent this also parallels the Halving algorithm and the linear programming results of Maass and Turan. In those algorithms, the current hypothesis is chosen so that any counterexample (mistake/adjustment) will eliminate a large portion of the remaining hypothesis space. If any counterexample to any of the remaining hypothesis could be chosen, learning time could be exponential rather than polynomial and adjustments could in fact be made on all 2^d points.

As previously observed, results for binary features can be generalized to n-valued features. In particular, the AND/OR function can be generalized to require weights of size approximately n^{d-1}, and the lower bound on weight size for average LTU functions can be generalized to $n^{(d-1)/2}$ (Hampson & Volper, 1990). Based on this, the performance of the Basic algorithm generalizes from about 2^{2d} (using Muroga's function) to about n^{2d} (using the n-valued AND/OR function). The maximum number of LTU functions generalizes from 2^{d^2} to n^{d^2}, so the Halving algorithm will require at most $log_2 n^{d^2} = d^2log(n)$ points to distinguish all LTU functions. Maass and Turans's analysis generalizes from between d^2 and $d^2log(d)$ to between $d^2log(n)$ and $d^2(log(d) + log(n))$.

There are too many variations of input representation, input order, function type and learning algorithms to consider all combinations in detail here, so only a few especially relevant results will be summarized. As expected, Basic's performance under any circumstances are on the order of n^{2d}. On the other hand, empirical results of the Reflect algorithm with n-valued input in the range (-1, 1) and ShuffleCycle or FixedOrder input are roughly in agreement with the optimum analysis: for fixed d, adjusts were less than linear with n, but appeared to be slightly greater than the optimum $log(n)$. As before, Reflect with LeastCorrect order was similar to the performance of the Basic algorithm, i.e. about n^{2d}. These results were true for both the AND/OR and random functions. All tests were run with $d < 6$ and varying n, and in that range the AND/OR function is harder than random functions. AND/OR produced results reliably close to n^{2d} when LeastCorrect order was used, while results for random functions were better characterized with a somewhat lower-order polynomial (less than the $2d$ of the AND/OR function). The MinAdj algorithm was comparable, although slightly worse than the Reflect algorithm when using

ShuffleCycle or FixedOrder, and distinctly worse with other orderings.

In summary, it appears that under reasonable training conditions Reflect and MinAdj performance is close to the theoretical upper bound on optimum performance of $d^2(log(d)+log(n))$. Since this only differs from the lower bound by a factor of $log(d)$, this is remarkably close to optimum over all possible algorithms. However, since Reflect and MinAdj appear to be exponential in d using LeastCorrect order, such time complexity results are clearly only relevant within the context of particular presentation strategies.

7 Irrelevant features and the Winnow algorithm

Besides its growth rate with the number of relevant features, d, another important question is how well Reflect deals with irrelevant features. The upper bound on Basic adjustments increases linearly with the number of irrelevant features since neither W nor a is changed, and M increases linearly. Empirically, both Basic and Reflect adjustments increase linearly. This contrasts with the Winnow algorithm for training LTUs (Littlestone, 1988) which is logarithmic in the number of irrelevant features. This can be a considerable advantage when there are many features, but only a few relevant ones. In addition, the Winnow algorithm is of interest because of its unique, multiplicative weight adjustment strategy, as contrasted with the additive update rule of Basic and Reflect. That is, while Basic and Reflect add or subtract a potentially adjustable amount from the weights in order to correct the output for a given input pattern, Winnow multiplies or divides the weights by an adjustable, but predetermined learning rate factor, $\alpha > 1$. Specifically, if output is too low, the weights of all present features are multiplied by α, and if output is too high, they are divided by α. The threshold is any fixed, positive number, t; it is not adjusted during training.

$$\begin{array}{l} \text{if } W \cdot F - t <= 0 \text{ and } F \text{ is a positive example} \\ \qquad \text{then } W_i \leftarrow \alpha * W_i \text{ (for each } F_i = 1) \\ \text{if } W \cdot F - t >= 0 \text{ and } F \text{ is a negative example} \\ \qquad \text{then } W_i \leftarrow 1/\alpha * W_i \text{ (for each } F_i = 1) \end{array}$$

In order to implement this learning strategy, feature weights must always be positive. For our experiments, the feature weights are initialized to 1.0 and the threshold is fixed at d.

Despite its superior time complexity on irrelevant features, the Winnow algorithm has some drawbacks as a general LTU learning algorithm. Specifically, the appropriate setting of the learning rate factor depends on the complexity of the function being learned. If it is set too high, the algorithm does not converge, and if it is set too low, learning can be slower than Basic. When learning random LTU functions, the rate factor must shrink exponentially toward 1.0 with the number of relevant features. In addition, because all feature weights must be positive, in order to represent functions with negative weights, the number of input features has to be doubled (since a positive weight on feature absence is equivalent to a negative weight on feature presence). Since half the new features can be considered irrelevant, this is not in itself prohibitively expensive, but it requires that the learning rate be reduced, which further reduces performance. Winnow, even when correctly tuned, is much slower than Reflect when learning arbitrary LTU functions of relevant features. For example, with the learning rate set low enough to learn 100 random functions of 12 features using ShuffleCycle order, Winnow took thousands of cycles and tens of thousands of adjusts, which is not much better than the Basic algorithm. On the same set of functions, Reflect took about 4 cycles and 100 adjustments (see Table 4).

Interestingly, the Reflection algorithm's output adjustment strategy of "reflecting" output across 0 on each adjustment can also be applied to the Winnow algorithm. For binary features the adaptation is straightforward: namely replace the fixed multiplicative constant α by an adaptive one according to the following rule:

$$\alpha = (2t - F \cdot W)/(F \cdot W)$$
$$W_i \leftarrow \alpha * W_i \text{ (for each } F_i = 1)$$

However two minor issues must be addressed: what to do when the output is too small to reflect accurately, and the fact that, because all weights must be positive, if $F \cdot W$ is more than twice t, it can't be reflected. In both cases a single, fixed learning rate factor of 1.1 or 1/1.1 was used. Any positive value can be used for the threshold, but for good results over a wide range of both relevant and irrelevant features a value around 20 or 30 is reasonable. Probably an adaptive threshold will be better, but that is another research issue. This use of the Reflection output adjustment strategy eliminates the need to tune the Winnow learning rate, and presumably also reduces over-generalization. For the above test of 12-feature random functions, this dramatically improved the performance of Winnow to almost match Reflect. Using the same algorithm (two weights per features plus output reflection), results were also improved for the AND/OR and Muroga's function. These two function require only positive feature weights and thus do not require a doubling of the number of features, but results were much better when the features were doubled, again approaching, but not equaling, the performance of the Reflection algorithm. As with Reflection, results with FixedOrder were comparable to ShuffleCycle, LongAdj was generally best and LeastCorrect was much worse. In fact, under some conditions, the algorithm failed to converge with LeastCorrect order, a point that needs to be scrutinized in future work. While the results were similar to Reflection with regard to relevant features, the growth with the number of irrelevant features was less than linear, making this new Winnow/Reflect algorithm an interesting approach in its own right.

It is not our goal to investigate this new hybrid algorithm in this paper, that is the topic of future research. The main point is that the general "output reflection" strategy appears to be equally applicable and beneficial to both Basic (additive) and Winnow (multiplicative) weight adjustment strategies. In addition, a MinAdj version of Winnow produced performance comparable to the Winnow/Reflect algorithm, suggesting that the performance of Winnow, like Basic, is limited by problems of over-generalization.

8 Boundary Patterns

The boundary patterns for a linearly separable classification are those that, if an LTU is correct on them, it is correct on all patterns. This provides a different perspective on LTU training since it focuses on the number of different patterns misclassified rather than the total number of misclassifications.

LeastCorrect presentation order trains *only* on the boundary patterns meaning that while it makes the greatest number of mistakes, it misses on the fewest different patterns. (It is possible to set the weight vector so that least-correct ordering will adjust on a non-boundary pattern, but this does not appear to happen in actual practice.) It is an interesting fact that known "hard" LTU functions (those that require large weights) have few boundary patterns. For example, the AND/OR function has only $d+1$ boundary patterns both theoretically and as observed in LeastCorrect ordering (Volper & Hampson, 1987; Anthony *et al*, 1992). Random LTU functions have an average of $2d$ boundary patterns both theoretically (Cover, 1965) and as determined by LeastCorrect training (Hampson, 1991).

Based on this result, an LTU learning algorithm that never missed on the same pattern twice would be expected to make an average of $2d$ mistakes on random LTU functions if inputs were presented in LeastCorrect order. Like the Basic algorithm, Reflect misses on about $2d$ different patterns for random functions, but although it makes fewer errors, it still misses on the same patterns an exponential number of times.

Using Shuffle order, both Basic and Reflect miss on a greater number of different patterns, but on the average, Reflect doesn't miss on the same pattern more than twice while Basic can still misclassify the same pattern thousands of times.

It is possible to specify most LTUs by giving the classification of approximately $2d$ points, but some functions, such as (at least $d/2$ of d), which has exactly

$$\binom{\mathrm{d}}{d/2-1} + \binom{\mathrm{d}}{d/2}$$

boundary patterns, have on the order of 2^d boundary patterns. Since all of these points have to be seen to completely define the function, this can only be reconciled with the $d^2log(d)$ upper bound by the fact that a smart algorithm will correctly guess the function long before it has seen enough of the boundary patterns to actually define it. Both Basic and Reflect have this property since they learn the (at least x of d) function in polynomial time. Required weight size of the (at least x of d) function is linear with d which guarantees polynomial Basic learning time (Hampson & Volper 1986).

9 Discussion

Maass and Turan (1994) describe a feasible $d^2(log(d) + log(n))$ LTU training algorithm based on linear programming methods but leave as an open question whether perceptron-like algorithms can produce comparable performance. Perceptron-like algorithms are of interest because, besides being much simpler, they are generally incremental, have minimal memory requirements and can track changing concepts and changing feature sets. They can also tolerate a certain amount of noise in the classification and/or input signals. Their most serious drawback has been that they appeared to be inherently slow, requiring exponential, rather than polynomial time to learn most LTU functions. This is trivially true for the Basic algorithm since the average LTU function of d features requires weights of exponential size, which in turn require an exponential number of adjustments. Even in the PAC learning model (Valiant, 1984) where the requirement for perfect classification is relaxed to permit approximately correct classification, learning time is still exponential (Baum, 1990; Schmitt, 1996). Only by further relaxing the requirement for input distribution independence in PAC learning was Baum (1990) able to derive a polynomial bound.

The Basic algorithm is unavoidably exponential for any input presentation strategy if perfect classification is required, but our experiments with the Reflection algorithm indicates that something close to $d^2(log(d) + log(n))$ performance can in fact be achieved by perceptron-like algorithms under realistic training conditions. Input presentation orders exist under which Reflect acts more like the Basic algorithm (approximately n^{2d}), but on the whole, it is easier to get good than bad performance. This corresponds to Baum's (1990) observation that a uniform input distribution is apt to give good results because it does not preferentially select boundary patterns. FixedOrder or ShuffleCycle presentation give good results and are natural presentation strategies. It remains an open question whether there are perceptron-like algorithms that give good results on all orderings.

Although empirically successful, the Reflection algorithm leaves a number of unresolved issues. For example, it is not clear what the best strategy for avoiding, detecting or correcting roundoff error is. Increasing the shrinkage rate in adjustment size by reducing λ generally increases learning speed, but also increases the probability of roundoff errors which slow learning. Increasing λ as in Reflect 1 has the opposite effect. The optimum tradeoff between the two effects is situation dependent. The "shrink and repair"approach of Reflect 2 is simple, but is also unlikely to be optimal.

Another important area of possible improvement is the linear increase in learning time with the number of irrelevant features, rather than logarithmic as with Winnow. The combined Winnow/Reflect algorithm offers hope of combining the polynomial learning speed of Reflect on relevant features with the logarithmic learning speed of Winnow on irrelevant ones.

Finally, there is the issue of accuracy. The performance of both Basic and Reflect is degraded by noise or non-separable points, but because it makes large adjustments in response to large errors, the accuracy of Reflect is affected more. The use of model averaging greatly improves the accuracy of both algorithms but Basic results are still often better under these conditions.

One obvious limitation of LTUs is that most functions are not linear, although a surprising number of real-world classification problems can at least be approximated that way. One standard extension of the LTU model is to add higher-order terms to the original feature set. A recent and very successful approach to adding higher-order terms is given by Cortes and Vapnik (1995). This maintains the simplicity of LTU learning while expanding the class of learnable functions. With perceptron-like algorithms, the feature set can be expanded incrementally. Given its training speed and the small number of cycles to convergence (or conversely to detect non-convergence), the Reflection algorithm may be ideally suited for this sort of approach.

In order to better understand the behavior of the Reflect algorithm, the MinAdj strategy was tested. Although not identical, the results were similar, showing the same $d^2(log(d) + log(n))$ performance with ShuffleCycle and Fixed ordering. These and other results suggest that over-generalization is a significant rate-limiting factor of the Basic algorithm, although it is still only a conjecture that the performance of Reflection can be interpreted in terms of reduced over-generalization. Output-reflection and MinAdj versions of Winnow also dramatically improved performance for that algorithm suggesting that over-generalization is a significant problem for Winnow as well. The fact that MinAdj is not uniformly superior to Reflection indicates that there are cases when over-generalization can be useful, but on the average it appears to be quite detrimental. Few people are apt to be surprised that extrapolation beyond the region of available data can be detrimental, but it is still rather dramatic that its elimination from a commonly-used algorithm (Basic Perceptron) would convert n^{2d} behavior to approximately $d^2(log(d)+log(n))$ under normal training conditions.

10 References

Agmon, S. (1954) The Relaxation Method for Linear Equalities. *Canadian Journal of Math. 6*, 382–392.

Anthony, M., Graham, B., Cohen, D., Shawe-Taylor, J. (1992) On exact specification by examples. *Fifth Annual ACM Workshop on Computational Learning Theory.* ACM Press, New York, 311-318

Baum, E.B. (1990) The perceptron algorithm is fast for nonmalicious distributions. *Neural*

Computation 2, 248–260.

Cortes, C. & Vapnik, V. (1995) Support-Vector Networks. *Machine Learning 20*, 273–297

Cover, T.M. (1965) Geometric and statistical properties of systems of linear inequalities with application in pattern recognition. *IEEE Trans. Elec. Comp.*, EC-14, 326–334.

Duda, R.O. & Hart, P.E. (1973) *Pattern Classification and Scene Analysis.* New York: John Wiley & Sons.

Hampson, S.E. (1991) Generalization and Specialization in Artificial Neural Networks. *Progress in Neurobiology 37*, 383–431.

Hampson, S.E. & Kibler, D. (1996) Large Plateaus and Plateau Search in Boolean Satisfiability Problems: When to Give Up Searching and Start Again, *Proceedings of American Mathematical Society*, Special DIMACS Issue, 437–455.

Hampson, S.E. & Volper, D. J. (1986) Linear Function Neurons: Structure and Training. *Biol. Cyber. 53*, 203–217.

Hampson, S.E. & Volper, D.J. (1990) Representing and Learning Boolean functions of multivalued features. *IEEE Transactions on Systems, Man and Cybernetics 20*, 67–80.

Hastad, J. (1994) On the Size of Weights for Threshold Gates. *SIAM Journal on Discrete Mathematics 7*, 484–492.

Lewis II, P.M. (1966) A lower bound on the number of corrections required for convergence of the simple threshold gate adaptive procedure. *IEEE Trans. on Electronic Computers 15*, 933–935.

Littlestone, N. (1988) Learning quickly when irrelevant attribute abound: A new linear-threshold algorithm. *Machine Learning 2*, 285–318.

Maass, W. (1994) Perspectives of Current Research about the Complexity of Learning on Neural Nets. In *Theoretical Advances in Neural Computations and Learning.* Eds. Roychowdhury, V. Siu, K. Orlitsky, A. Boston: Kluwer Academic Publishing. Boston, MA.

Maass, W. & Turan, G. (1994) How Fast Can a Threshold Gate Learn. In *Computational Learning Theory and Natural Learning Systems.* Eds. Hanson S.J., Drastal G.A., Rivest R.L. MIT Press. Cambridge, MA

Minsky, M. & Papert, S. (1969) *Perceptrons.* Cambridge, MA: MIT Press.

Motzkin, T.S. & Schoenberg, I.J. (1954) The Relaxation Method for Linear Equalities. *Canadian Journal of Math. 6*, 393–404.

Muroga, S. (1971). *Threshold Logic and Its Applications.* New York: John Wiley and Sons.

Nilsson, N.J. (1965) *Learning Machines.* New York: NcGraw-Hill.

Rosenblatt F. (1958) The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 386–408.

Schmitt M., (1996) Lower Bounds on Identification Criteria for Perceptron-like Learning Rules. In: *Proceedings of the thirteenth European meeting on cybernetics and systems research.* EMCSR'96 2, 1049–1054, Ed. Trappi, R., Vienna.

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

Volper, D.J. & Hampson, S.E. (1987) Learning and Using Specific Instances. *Biol. Cyber. 57*, 57–71.