Learning with Limited Numerical Precision Using the Cascade-Correlation Algorithm

Markus Hoehfeld and **Scott E. Fahlman**

May 3, 1991

CMU-CS-91-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A key question in the design of specialized hardware for simulation of neural networks is whether fixed-point arithmetic of limited numerical precision can be used with existing learning algorithms. We present an empirical study of the effects of limited precision in Cascade-Correlation networks on three different learning problems. We show that learning can fail abruptly as the precision of network weights or weight-update calculations is reduced below 12 bits. We introduce techniques for dynamic rescaling and probabilistic rounding that allow reliable convergence down to 6 bits of precision, with only a gradual reduction in the quality of the solutions.

This research was supported by Siemens Corporation. The experiments described here were conducted during a visit by Markus Hoehfeld to Carnegie Mellon in the fall of 1990. Markus Hoehfeld's permanent address is Siemens AG, ZFE IS INF 2, Otto-Hahn-Ring 6, W-8000 Munich 83, Germany.

Keywords: Artificial Intelligence, Learning, Neural networks, Cascade-Correlation, Processor Architectures

1. Introduction

A key question in the design of specialized high-speed neural network hardware is whether limited-precision arithmetic can be used successfully in conjunction with existing learning architectures and algorithms. Most neural network simulations on conventional computers have been done using a 32-bit floating-point representation for the network's weights, unit activations, and weight-update calculations. If short fixed-point numbers can be used instead, it will be possible to put more weights and processing elements on a chip, with a corresponding increase in the ratio of speed and capacity to hardware cost.

Until recently, there has been little reliable information on the effects of limited precision on neural net learning algorithms. A number of theoretical models have shown that networks can compute various functions with limited-precision or even binary weights, but this says nothing about the requirements of the learning process. Recently, a few careful empirical studies have been published for networks using the back-propagation learning algorithm: [Baker, 1988] and [Hollis, 1990] both report a sudden, sharp failure of learning capability when the precision of the stored weights is reduced below some critical level—about 12 bits[1] in the problems they studied.

In a more recent study, [Holt, 1990] developed a statistical model of precision, rounding, and truncation that predicted a collapse of learning ability at around 12 to 16 bits of precision, depending on the details of the problem. The predicted collapse was then observed in simulations. The authors emphasize that this is a failure of learning, not of performance: once a useful set of weights has been learned, a machine of 8-bit precision or less may be adequate to simulate the network's performance.

In this paper, we present an empirical study of the effects of limited numerical precision in networks employing the Cascade-Correlation (or "CC") supervised learning algorithm developed by Fahlman and Lebiere[Fahlman, 1990]. CC is an example of a *constructive* learning algorithm: it begins with a minimal network and adds new hidden units as needed during training. Public domain simulation code for CC has been available since early 1990, and the community of CC users has been growing steadily since that time. The number of CC users is still small compared to the number of back-propagation users. However, we believe that this study will be of general interest for several reasons:

- The very difficult learning tasks that might require fast parallel hardware are the same ones that would benefit the most from the fast learning and good scaling properties of CC.
- We demonstrate that the most significant problem caused by limited precision in CC is one that will occur in all gradient-following algorithms: because of quantization of the weights and weight-updates, areas with small error gradients appear to be flat plateaus.
- We describe techniques for dynamic rescaling and probabilistic rounding that make it possible for CC networks to converge reliably, even when the precision is reduced to 6 bits. These same techniques are applicable to other learning architectures that follow an error gradient.
- Because CC adds new hidden units as needed during learning, it can sometimes compensate for limited precision by creating a few additional units. This can lead to a graceful degradation in learning performance, rather than abrupt failure. We will describe the conditions under which such graceful degradation occurs.

[1] Throughout this paper, when we speak of n bits of precision, this is the word length not including any sign bit.

- CC uses the quickprop algorithm[Fahlman, 1988] instead of simple, linear gradient descent to update the weights in the network. Quickprop uses information about the second derivative of the error to speed up convergence. We demonstrate that such second-order methods do not experience any special problems due to limited precision.

These issues were studied empirically. Simulations were run on three well-known learning problems: 6-bit parity, the "mines vs. rocks" sonar classification task[Gorman, 1988], and the two-spirals task[Lang, 1988]. These simulations used a modified version of the cascade-correlation simulator that allows us to independently control the precision of the stored weights, the sigmoid function, and the weight-update calculations.

2. Review of Cascade-Correlation

Cascade-Correlation[Fahlman, 1990] is a supervised learning architecture that builds a near-minimal multilayer network topology in the course of training. Initially the network contains only inputs, output units, and the connections between them. This single layer of connections is trained (using the quickprop algorithm[Fahlman, 1988]) to minimize the error. When no further improvement is seen in the level of error, the network's performance is evaluated. If the error is small enough, we stop. Otherwise we add a new hidden unit to the network in an attempt to reduce the residual error.

To create a new hidden unit, we begin with a pool of *candidate units*, each of which receives weighted connections from the network's inputs and from any hidden units already present in the net. The outputs of these candidate units are not yet connected into the active network. Multiple passes through the training set are run, and each candidate unit adjusts its incoming weights to maximize the correlation between its output and the residual error in the active net. When the correlation scores stop improving, we choose the best candidate, freeze its incoming weights, and add it to the network. This process is called "tenure." After tenure, a unit becomes a permanent new feature detector in the net. We then re-train all the weights going to the output units, including those from the new hidden unit. This process of adding a new hidden unit and re-training the output layer is repeated until the error is negligible or we give up. Since the new hidden unit receives connections from the old ones, each hidden unit effectively adds a new layer to the net (see Figure 1).

CC eliminates the need for the user to guess in advance the network's size, depth, and topology. A reasonably small (though not necessarily minimal) network is built automatically. Because a hidden-unit feature detector, once built, is never altered or cannibalized, the network can be trained incrementally. A large data set can be broken up into smaller "lessons," and feature-building will be cumulative.

CC learns much faster than standard back-propagation for several reasons: First only a single layer of weights is being trained at any given time. There is never any need to propagate error information backwards through the connections, and we avoid the dramatic slowdown that is typical when training backprop nets with many layers. Second, this is a "greedy" algorithm: each new unit grabs as much of the remaining error as it can. In a standard backprop net, all the hidden units are changing at once, competing for the various jobs that must be done—a slow and sometimes unreliable process.

CC is particularly well-suited for implementation on parallel hardware with limited inter-processor communication. During the candidate-training phase (where the most time is spent for problems of any complexity), each of the candidate units receives the same set of inputs and the same error signal. Each candidate hill-

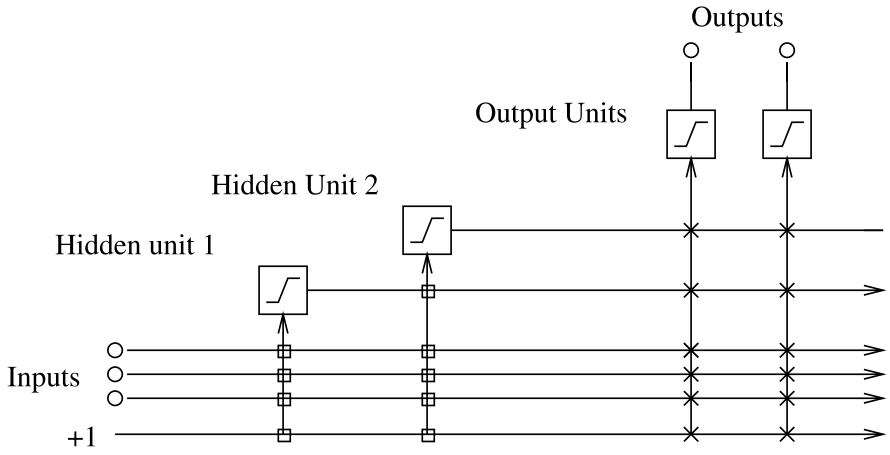

Figure 1: The Cascade-Correlation architecture after two hidden units have been added. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly.

climbs independently to maximize the correlation between its output and the residual error. These units do not need to communicate with one another except to pick a winning unit for installation into the active net.

We will now consider the arithmetic operations that occur during CC learning and that are affected by limited numerical precision. Output units are trained to minimize the familiar sum-squared error measure

$$E = \frac{1}{2} \sum_{o,p} (y_{op} - t_{op})^2 \tag{1}$$

where y_{op} is the observed value of output o for training pattern p, and t_{op} is the desired or target output. E is minimized by gradient descent using

$$e_{op} = (y_{op} - t_{op}) f'_p \qquad \text{and} \tag{2}$$

$$\frac{\partial E}{\partial w_{oi}} = \sum_p e_{op} I_{ip} \tag{3}$$

where f'_p is the derivative of the sigmoid activation function of the output unit for pattern p, I_{ip} is the value of the input (or hidden unit) i, and w_{oi} is the weight connecting input i to output unit o.

Candidate units are trained to maximize C, the correlation between the candidate unit's output y and the residual errors e_o still observed at the outputs of the active network. This correlation is computed over all the training patterns p. C is defined as

$$C = \sum_o \left| \sum_p (y_p - \overline{y}) (e_{op} - \overline{e_o}) \right| \tag{4}$$

where $\overline{y}$ and $\overline{e_o}$ are averages of y and e_o over all patterns p. The maximization of C proceeds by gradient ascent using

$$\delta_p = \sum_o \sigma_o (e_{op} - \overline{e_o}) f'_p \qquad \text{and} \tag{5}$$

$$\frac{\partial C}{\partial w_i} = \sum_p \delta_p I_{ip} \tag{6}$$

where σ_o is the sign of the correlation between the candidate unit's value and the residual error at output o. Each candidate unit in the pool starts from a different set of initial weights and independently tries to maximize its own C value.

Given the gradient values computed above, we could in principle minimize E and maximize C by simple gradient descent (or ascent). Instead, to get good convergence in a reasonable number of cycles, we use the quickprop algorithm to compute the updates. Abbreviating the slopes $\partial E/\partial w$ and $-\partial C/\partial w$ by S, respectively, the weight change Δw is computed by

$$\Delta w_t = \begin{cases} \epsilon\, S(t), & \text{if } \Delta w_{t-1} = 0 \\ \frac{S(t)}{S(t-1) - S(t)}\, \Delta w_{t-1}, & \text{if } \Delta w_{t-1} \neq 0 \text{ and } \frac{S(t)}{S(t-1) - S(t)} < \mu \\ \mu \Delta w(t-1), & \text{otherwise.} \end{cases} \tag{7}$$

Here, ϵ is a parameter that controls the linear steps used to get the algorithm started, and μ is a parameter that controls the maximum step-size, compared to the previous step. Weights are now updated using

$$w(t+1) = w(t) + \Delta w(t) \tag{8}$$

This description of quickprop is slightly over-simplified, but is sufficient for the purposes of this paper. See [Fahlman, 1988] for a more complete description.

3. Overall Experimental Design

A floating-point representation can represent numbers whose magnitude is very large or very small. The error introduced by roundoff or truncation is never more than a small fraction of the number's magnitude. A fixed-point representation, on the other hand, limits the range of magnitudes and introduces a constant minimum step-size between one representable number and the next. Given a fixed word-length of n bits, there is a trade-off between range and the minimum step-size: we can use the available bits to represent large numbers with coarse steps or a smaller range of numbers with finer steps. In a neural-net simulator, this choice must be made carefully: if the total range is too small, our simulator will suffer from overflow or clipping of intermediate values; if the total range is too large, there is a danger that small weight adjustments will fall below the minimum step-size and be lost.

For this study we developed a Cascade-Correlation simulator that uses floating-point numbers internally, but that models the effects of limited precision through application of the *precision function*, $P(x, n)$, at appropriate points in the code. n is the number of bits in the simulated word length, not counting the sign bit.

$$P(x,n) = \begin{cases} \text{if } x \leq -1, & -1 \\ \text{if } x \geq 1, & 1 - 2^{-n} \\ \text{otherwise}, & x \text{ rounded down to multiple of } 2^{-n} \end{cases}$$

Thus, P simultaneously implements the two kinds of *precision error* introduced by a fixed-point representation: *clipping* of the real value to fit within a certain range of values—in this case, the range -1 to

+1—and *truncation* or *quantization* errors due to the limited word-length. To reduce notational clutter, we will generally treat the parameter n as implicit and simply write $P(x)$.

Cascade Correlation may be affected by limited precision in three different places: in the representation of weights in the network, in the computation of the sigmoid function, and in the accuracy of the arithmetic operations used in computing the slopes, correlations, and weight updates during learning. We will refer to these effects as *weight precision*, *sigmoid precision*, and *weight-update precision*, respectively. Our simulator allows us to study these effects individually or in any combination.

We chose three very different problems for our simulation studies. The first of these is 6-bit parity. With standard CC learning, this problem normally requires the construction of 3 or 4 hidden units for correct classification of all 64 input patterns.

The second problem is is a sonar signal classification problem, mines vs. rocks[Gorman, 1988]. The network is trained on 104 patterns, each of which has 60 analog inputs and one binary output. This problem is linearly separable, which means it can be solved using the input-to-output weights alone, without the creation of any hidden units. However, even with quickprop learning, it can take a long time (2000 or more passes through the training set) to find a perfect solution with no hidden units. With the usual parameter settings, CC will normally create one hidden unit, which makes the solution much easier.

The third problem is the "two-spirals" problem, originally proposed by Alexis Wieland as a very difficult learning benchmark for back-propagation and related learning architectures, and later studied by [Lang, 1988] and [Fahlman, 1990]. This problem has two analog inputs, a single binary output, and a set of 194 training points arranged in such a way that a very complex separating surface must be created. With standard CC, this problem typically requires the creation of 11-15 hidden units.

Unless otherwise specified, all of the simulation results given in this paper are averages over five runs with different random initial weights. In addition to the number of hidden units built, we list the average number of training epochs required. (An *epoch* is defined as one pass through the entire training set, followed by a weight update.) In cases where some of the runs failed to converge after building 30 hidden units, we indicate the number of successful trials with a superscript in the table entry; in these cases, the number of units and epochs reported is an average over the successful cases only.

To keep results comparable across all our experiments, we did not optimize the learning parameters separately for each problem. For the output-training phase, we used $\mu = 2.0$ and $\epsilon = 0.35$; for the candidate-training phase we used $\mu = 2.0$ and $\epsilon = 1.0$. The epsilon values were divided by the number of patterns in the training set. Since all of the problems had binary outputs, training was considered complete when the target values of ± 0.5 were approximated to within 0.35. Each training phase was terminated after there had been been no significant improvement for 40 epochs or after a maximum of 400 epochs. Because of these conservative parameter choices, the number of hidden units should be considered the primary measure of learning performance; the epoch counts could be considerably improved with a bit of parameter tuning.

In all cases, we use the symmetric sigmoid activation function, with a range of [-0.5,0.5], for both the output and hidden (or candidate) units:

$$f(x) = \frac{1}{1 + e^{-x}} - 0.5 \tag{9}$$

An offset of 0.1 was added to the derivative f' at output units (only) to eliminate the "stuck units" problem (see [Fahlman, 1988]).

	$\pm\infty$	±128	±64	±32	±16	±8	±4	±2	±1 (var λ_u)
6-Parity	3.8			3.8	3.8	4.2	7.2	12.4	4.4
	838			831	836	740	1125	1758	785
Sonar	1.0	1.0	1.2	1.4	1.8	2.0	3.4	6.4	1.0
	654	659	875	1044	1126	1224	1329	1689	837
Spirals	12.6	11.4	12.2	13.6	17.2	18.4	27.2	fail	12.6
	3881	3528	3561	3720	4566	4393	5369		3706

Table 1: Number of hidden units and learning times for floating-point weights clipped to various range limits during training. The last column shows results for a weight range of ±1 when the unit's gain (λ_u) is adjusted automatically during training.

4. Limited Weight Range and Variable Gain

Given the above definition of *P*, we would like to minimize clipping errors by choosing a representation that keeps the network's weights in the range [-1, +1]. This is achieved by the use of a common normalization factor λ. For example, instead of using weights w in the range [-8, +8], we instead use weights w' in the range [-1, +1], with $\lambda = 8$. The "true" value w of a weight is then given by $w = \lambda w'$. Note that instead of multiplying every weight by λ, we can implement this scaling more efficiently as a *variable gain* term in the sigmoid calculations:

$$f(x) = \frac{1}{1 + e^{-\lambda x}} - 0.5 \qquad (10)$$

The problem of choosing an appropriate λ is equivalent to the problem of choosing a sufficient weight range. [Baker, 1988] report that 3 bits before the binary point—corresponding to a weight range of [-8,+8] or $\lambda = 8$—were sufficient for their pattern recognition application, but they correctly conjectured that this parameter depends on the application.

To explore the effects of limited weight range by itself, we ran an initial set of simulations using floating-point arithmetic during learning, but with weights clipped at different range limits. The weight range was varied from ±128 down to ±2. Table 1 shows the number of hidden units created and installed by CC for different weight ranges, along with the learning times measured in epochs. For comparison we include results for unbounded weights, ($\pm\infty$). The rightmost column will be explained below.

As the range of the weights is decreased below a certain value, different for each problem, the learning performance of CC begins to deteriorate. However, CC does not fail abruptly when the weight-range is too small. Instead, it attempts to compensate for limited weight-range by building more than the usual number of hidden units (with a corresponding increase in learning time). Sonar and parity can still be learned with weights clipped to ±2, though the solution requires 4-6 times as many units as in the unclipped case. The only outright failure was for the difficult two-spirals problem with clipping at ±2.

Since it is difficult to decide in advance what the weight-range should be for a given problem, and since the magnitude of the weights can change as training progresses, we investigated a scheme in which the gain λ is adjusted automatically. A separate λ_u is associated with each unit. Initially, the weights are set to random values in the range [-1,+1], and all of the λ values to 1. As the weights grow or shrink during training, we adjust λ_u upwards or downwards so that the incoming weights usually fit into the range [-1,+1]. This eliminates most problems due to clipping. This kind of approach is sometimes called a *block-floating-point* representation: it is similar to the use of a floating-point representation, in which each number carries its own scale factor, but in this case we use a single scale factor for a whole set of weights.

We use the following simple formula to modify λ_u after each round of weight updates:

$$\lambda_{new} = \begin{cases} \lambda_{old} * 2 & \text{if } \overline{w} > 0.5 \\ \lambda_{old}/2 & \text{if } \overline{w} < 0.25 \\ \lambda_{old} & \text{otherwise} \end{cases} \qquad w_{new} = \frac{\lambda_{old}}{\lambda_{new}} w_{old}$$

Here, the mean $\overline{w}$ over a unit's incoming weights is computed only for weights larger than 0.1. This is done to reduce the influence of unimportant connections whose weights are decaying toward zero. We tested a number of ways to change λ_u more smoothly, but without significantly better results.

This use of variable gain has several benefits:

- We no longer need to guess a static value of λ that will be appropriate for the current problem.
- Different units and layers in the network may have radically different values of λ_u.
- At any given time, we make efficient use of the available word-length to represent the current weight values. No bits are wasted in anticipation of larger weights, which might or might not evolve in the course of learning.

The implementation of [Hollis, 1990] also uses a gain term or scaling factor to prevent clipping of weights during learning. However, this scale factor is not automatically updated during training; an appropriate value must be chosen in advance, based on past experience.

The results for learning with variable λ_u are shown in the rightmost column of table 1. With variable gain, a fixed-point representation performs as well as floating point on the sonar and two-spirals problems, and nearly as well for parity. Solutions for the parity problem generally develop large ratios between significant weight values, and some clipping is observed in this problem as fast-growing weights move far away from the average for that unit. It is interesting to note that in most of the trials we ran the output unit's weights (after multiplication by λ) grew to about the same size that they reached without any clipping.

The limited-precision simulations described in the remainder of this paper all employ this variable gain technique. Similarly, the error terms e_{po} and the derivatives $\partial E/\partial w$ and $\partial C/\partial w$ were scaled by variable global factors to keep these values within reasonable ranges. These factors were kept constant for an entire training epoch. Scaling is applied before the application of the precision function in all cases. The ϵ parameter was scaled inversely to the gain λ_u for the unit whose weights are being updated. This prevents excessively large steps at high gains.

5. Effects of Limited Weight Precision

To investigate the effects of limited precision in the weight values, we begin with random weights $P(r)$ instead of r. The weight update formula (8) is changed to

$$w(t+1) = P(w + \Delta w) \tag{11}$$

	∞	$\infty^{\pm 1}$	15	14	12	10
6-Parity	6.0	4.4	5.8	4.8	4.8	fail
	1039	785	1035	899	842	
Sonar	1.6	1.0	1.2	1.2	1.8	fail
	1044	837	879	901	1058	
Spirals	12.4	13.0	12.6	13.0	fail	fail
	3702	3826	3520	3752		

Table 2: Number of hidden units for three selected tasks with limited weight precision

That is, no matter what Δw may be, the updated weight is again quantized. If the weight grows out of bounds, it is clipped by the P function.

We ran our three benchmark problems with the precision of the weights limited in this way. Results for various choices of n are shown in table 2. For comparison, we also show the performance for unrestricted floating-point weights (denoted by ∞ in the table), and for floating-point weights with the range restricted to [-1,+1], denoted by $\infty^{\pm 1}$.

We see that the performance of CC is affected very little by a reduction in weight precision until, at a certain value of n, we get a sudden failure. This value of n depends on the problem: it is 12 for two-spirals and 10 for the other benchmarks. This result is not too surprising, since [Baker, 1988] and [Hollis, 1990] report that backpropagation learning fails abruptly for less than 12 bits of weight precision in the problems that they investigated.

The cause of this problem is clear: below the critical value of n, most of the weight updates are too small to push the weight to the next step. After P is applied to the weights, these updates have no effect at all. Both the candidate-training and output-training phases fail to perform properly under these conditions.

6. Effects of Limited Sigmoid Precision

To investigate the effects of a limited precision sigmoid function, equation (9) is changed to

$$f(x) = P\left(\frac{1}{1 + e^{-8\,P(\lambda x)}} - 0.5\right) \tag{12}$$

The factor of 8 here is used to alter the effective sharpness of the sigmoid, so that the [-1,+1] range of the precision-limited operand will extend well out into the nonlinear "saturated" portion of the sigmoid. The sigmoid can be computed directly or by table-lookup with $P(\lambda x)$ as an input.

In sigmoid-precision mode, the network's input patterns are also subject to the precision function P. Since these inputs participate in the same computations as the hidden-unit outputs, they must be represented in the same format and therefore will suffer the same limitations on their precision.

The results of these experiments are summarized in table 3. We see that training is affected very little by limited precision in the activation function. Convergence is uniform down to $n = 6$. There is a slight deterioration at $n = 4$, but one or two additional hidden units are sufficient to correct this problem.

	∞	$\infty^{\pm 1}$	12	10	8	6	4
6-Parity	3.4	3.8	4.8	4.2	3.8	4.2	4.6
	749	825	971	892	809	923	919
Sonar	1.0	1.0	1.0	1.0	1.0	1.0	2.0
	637	662	643	652	660	654	993
Spirals	13.0	12.2	11.8	13.2	12.6	13.0	15.0
	4061	3671	3661	3912	3791	3847	4116

Table 3: Number of hidden units for three selected tasks with limited sigmoid precision

7. Effects of Limited Weight-Update Precision

We next studied the effect of limited precision in the computation of the weight updates, Δw, for both the candidate units and the output units. These computations involve a sequence of arithmetic operations. Many of the intermediate values have to be stored temporarily in registers of fixed length, leading to a variety of precision errors. Formulae (2)- (7), which compute errors, correlations, derivatives and finally weight update terms, are changed as shown below to incorporate precision constraints. All multiplications involve only limited precision operands.

$$e_{op} = P((y_{op} - t_{op}) f'_p) \tag{13}$$

$$\partial E / \partial w_{oi} = P(\sum_p e_{op} I_{ip}) \tag{14}$$

$$C = P(\sum_o \left| P(\sum_p y_p e_{op}) \right|) \tag{15}$$

$$\delta_p = P(\sum_o \sigma_o (e_{op} - \overline{e_o}) f'_p) \tag{16}$$

$$\partial C / \partial w_i = P(\sum_p \delta_p I_{ip}) \tag{17}$$

$$\Delta w_t = \begin{cases} P(\epsilon S(t)), & \text{if } \Delta w_{t-1} = 0 \\ P\big(P(\frac{S(t)}{P(S(t-1) - S(t))}) \Delta w_{t-1}\big), & \text{if } \Delta w_{t-1} \neq 0, P(\frac{S(t)}{P(S(t-1) - S(t))}) < \mu \\ P(\mu \Delta w_{t-1}), & \text{otherwise.} \end{cases} \tag{18}$$

Note that the summations in these formulae use full precision and only the final accumulated results are rounded by P. The minimal required accumulator size is problem dependent. In the worst case, $2n + \log l$ bits are needed to accurately sum over l terms, where each term is the product of two n-bit numbers. The number of terms being summed depends on the size of the training set and the fan-in of the units in the network.

For neural network learning we can usually use a smaller accumulator than the above formula would indicate [Hoehfeld, 1990]. It is not necessary to use all $2n$ result bits of a $n * n$ multiplication in the subsequent accumulation. Furthermore, all accumulations sum over positive and negative terms and thus only need

	∞	$\infty^{\pm 1}$	15	14	12	10
6-Parity	4.4 885	4.0 870	3.8 841	3.6 832	4.4 866	fail
Sonar	1.0 657	1.0 720	1.0 735	1.0 846	2.0 1186	fail
Spirals	13.2 3897	12.4 3896	12.0 3914	12.5[4] 4115	fail	fail

Table 4: Number of hidden units for three selected tasks with limited weight-update precision.

about $\sqrt{l}$ overflow bits. These issues were not systematically addressed in this work because of the added complexity they introduce; we simply assume that the accumulators are big enough to do the job without overflowing.

Table 4 shows simulation results obtained when the weight update computations are restricted in this way. The results are about the same as those obtained for limited weight precision and also show about the same "cut-off" precision. Once again, we observe that beyond the cutoff point, most of the proposed weight updates have been rounded to zero and the learning stalls.

8. Probabilistic Weight Updates

As we saw in the previous sections, the straightforward implementation of limited precision fails abruptly for word lengths less than about 12 bits because most of the Δw are too small to push the weight w to the next legal value. Areas of weight-space which previously had a small gradient now appear to be flat plateaus.

One way around this problem is to always update a weight by at least the minimal step. However, on the problems we studied, this led to oscillations and the training failed to converge. Another possibility is to use *probabilistic weight updates*: whenever a proposed update Δw falls below the minimal step size, we take the minimal step with some probability p that is proportional to the size of Δw.

These *probabilistic weight updates* gave significantly better results for low precision. However, we still observed many wasted epochs in which no weight was updated because all of the Δw's were small. We therefore added the constant $1/\text{fan-in}$ to the probability to make it likely that at least one weight in each unit would be changed after every epoch. This resulted in much faster learning and did not cause severe oscillations.

From an implementation point of view, probabilistic weight updates turn out to be just a different rounding technique. The problem of vanishing update terms is due to the truncation of multiplication results. Whenever two numbers from [-1,1] are multiplied, the result is a smaller number and may fall below the minimum step size. It will then be truncated to 0 by P. To keep very small multiplication results — in a statistical sense — means to truncate those small numbers to either 0 or the quantization level 2^{-n}. Thus, *probabilistic truncation* is defined by

$$P_p(x) = \begin{cases} \text{if } |x| \geq 2^{-n} & P(x) \\ \text{if } |x| < 2^{-n} & \begin{cases} \sigma(x)\, 2^{-n} & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases} \end{cases} \tag{19}$$

where $\sigma(x)$ is the sign of x and the probability p is given by

$$p = \frac{1}{\text{fan-in}} + 2^n x \tag{20}$$

Probabilistic truncation doesn't help in a longer sequence of multiplications, because once an intermediate result is truncated to 0, the final result will be 0. Therefore, probabilistic truncation is only used in the last multiplication in a series. Any intermediate result whose magnitude is less than 2^{-n} is rounded away from zero. This is achieved by the sign-preserving precision function

$$P_s(x) = \begin{cases} \text{if } |x| \geq 2^{-n} & P(x) \\ \text{if } |x| < 2^{-n} & \sigma(x)\, 2^{-n} \end{cases} \tag{21}$$

The following changes were made to (11)-(18) to incorporate probabilistic truncation. The sign-preserving precision function was used to round the slopes $\partial E/\partial w_{oi}$ and $\partial C/\partial w_i$. Furthermore, the sign of the correlation with individual output units was preserved when truncating, because these signs are important in further slope computations.

$$\partial E/\partial w_{oi} = P_s(\sum_p e_{op} I_{ip}) \tag{22}$$

$$C = P(\sum_o \left| P_s(\sum_p y_p e_{op}) \right|) \tag{23}$$

$$\partial C/\partial w_i = P_s(\sum_p \delta_p I_{ip}) \tag{24}$$

The final weight update terms were truncated probabilistically according to

$$\Delta w_t = \begin{cases} P_p(\epsilon\, S(t)), & \text{if } \Delta w(t-1) = 0 \\ P_p\big(P(\frac{S(t)}{P(S(t-1) - S(t))})\Delta w_{t-1}\big), & \text{if } \Delta w_{t-1} \neq 0 \text{ and } P_p(\frac{S(t)}{P(S(t-1) - S(t))}) < \mu \\ P(\mu \Delta w_{t-1}), & \text{otherwise.} \end{cases} \tag{25}$$

Probabilistic truncation introduces a lot of noise into the learning procedure. Instead of decreasing steadily toward an asymptote, the error value oscillates. However, the low points of the oscillations decrease steadily toward an asymptote. This means that we can no longer use a simple quiescence test to end each training phase. Instead, we must record the best value seen so far in the training phase along with the weights that gave rise to that value. If no further improvement is seen after some time, we end the phase and revert to these stored weights, since they are better than the final weights. Some overhead can be eliminated by recording only significant improvements to the best weights.

	∞	$\infty^{\pm 1}$	12	10	8	6	4
6-Parity	6.0	4.4	4.8	5.0	4.6	6.2	6.8
	1039	785	828	909	836	1299	1327
Sonar	1.6	1.0	1.0	1.6	2.0	3.0	5.4
	1044	837	778	1060	1239	1721	2027
Spirals	12.4	13.0	13.0	12.2	13.8	15.0	20.7[3]
	3702	3826	3827	3411	3888	4464	5122

Table 5: Number of hidden units for three selected tasks with limited weight precision and probabilistic rounding.

9. Results Using Probabilistic Updates

Using probabilistic rounding we repeated the simulations for limited weight precision and for limited weight-update precision. (The limited-sigmoid case was not repeated because it did not suffer from steps being rounded to zero.) Table 5 shows the results for limited weight precision using probabilistic rounding. The performance for floating point (∞) and restricted-range floating point ($\infty^{\pm 1}$) are the same as in table 2, since there is no truncation in the "infinite" precision cases.

The results for 12 bits show considerable improvement over table 2, the simulations with ordinary truncation: for the sonar task, the number of hidden units decreases from 1.8 to 1.0 and for the spirals task, training with 12 bit now converges as well as unlimited precision, whereas it failed in table 2.

As precision is further reduced from 10 bits down to 6 bits, the performance of CC degrades gracefully. More hidden units are installed and learning time increases, but learning converges consistently even for 6 bits of weight precision. With only 4 bits of precision, the number of hidden units rises sharply. In these cases the steps become very coarse. If the λ value grows to, say, 128 the effective minimum step size is 8, and a single probabilistic decision can easily throw a unit from one extreme output value to the other. Under these conditions, even probabilistic updates are incapable of following a gradient.

Table 6 shows the performance of CC for limited precision in the weight-update computations using sign-preserving and probabilistic truncation. The results are qualitatively similar to those obtained for limited weight precision. Performance at 12 bits of precision is significantly improved over the simulation with normal rounding (table 4). Performance degrades slightly down to 6 bits of precision, and degrades dramatically at 4 bits.

Comparing table 5 to table 6, we see that the sonar task is more sensitive to limited weight precision, while the parity and spirals tasks are more sensitive to limited weight-update precision. This is easily explained: in the sonar task, most of the work is done by careful balancing of the weights in the output layer, so the weights themselves must be precise; the other two tasks do not require such precise weights in the ultimate solution, but they do require accurate gradient following to create an appropriate set of hidden units.

Finally, table 7 shows the results for probabilistic truncation, when weight, sigmoid, and weight-update precision are constrained to n bits simultaneously. The performance is comparable to the floating point results down to 12 bits of precision. In the range from 10 down to 6 bits, CC is able to compensate for limited precision by adding a few additional hidden units.

	∞	$\infty^{\pm 1}$	12	10	8	6	4
6-Parity	4.4	4.0	4.2	4.0	5.6	6.8	11.0[2]
	885	868	951	832	1087	1231	1978
Sonar	1.0	1.0	1.0	1.0	1.0	1.2	2.6
	625	760	786	767	823	951	1395
Spirals	12.0	12.8	12.0	12.6	13.8	15.6	fail
	3782	4156	3931	4103	4417	4013	

Table 6: Number of hidden units for three selected tasks with limited weight-update precision and probabilistic rounding.

	∞	$\infty^{\pm 1}$	12	10	8	6	4
6-Parity	4.4	4.4	4.6	5.6	5.4	4.8	10.8[4]
	858	837	856	1056	1038	1072	1561
Sonar	1.6	1.2	1.0	1.8	2.0	2.8	9.4
	987	906	859	1147	1263	1511	1980
Spirals	13.8	12.2	11.8	12.4	14.2	19.4	fail
	4248	3662	3308	3509	4067	4476	

Table 7: Number of hidden units for three selected tasks with weight, sigmoid, and weight-update precision simultaneously restricted to n bits, using probabilistic rounding.

It is interesting that solution quality and learning times for simultaneous precision constraints are roughly equal to the worst case observed for independent precision constraints. Thus the effects of precision errors do not add up, but the stronger requirements define the overall performance. The only exception to this observation is the 2-spirals task with 6 bits: around 15 hidden units are installed when weight precision or training precision are restricted independently, but 19 hidden units are built when both restrictions are present at once.

To summarize, the use of probabilistic rounding and variable gain significantly reduces the precision requirements for the Cascade-Correlation algorithm. At 12 bits of precision (plus sign), performance is equivalent to that obtained with a full 32-bit floating-point representation. Precision can be reduced to approximately 6 bits plus sign with, at worst, a moderate increase in learning time and the number of hidden units.

The probabilistic update technique becomes increasingly important as the precision is lowered. Table 8 lists the percentages of weight updates that fall below the minimum step size and thus are subject to probabilistic rounding. Over 90% of the weight updates are probabilistic for 8 bit precision and the ratio increases to around 95% for 6 bit. In the latter case nearly all weight updates are by the same step 2^{-n} and the weight-update computation merely provides the sign. This kind of learning was termed "coarse weight updates" in [Shoemaker, 1990], who report satisfactory results using coarse updates with back-propagation.

	12	10	8	6	4
6-Parity	59%	72%	84%	92%	98%
Sonar	71%	88%	95%	97%	99%
Spirals	67%	79%	89%	94%	96%

Table 8: Percentage of weight updates smaller than the minimum step size 2^{-n} for simultaneous precision constraints. These updates were rounded probabilistically to 0 or $\pm 2^{-n}$.

10. Quickprop vs. Simple Gradient Descent

As described in section 2, the Cascade-Correlation learning algorithm normally uses the quickprop algorithm for updating the weights once a gradient has been computed. Quickprop is used both in the output-training and candidate-training phases of learning to speed up convergence. It is possible to use ordinary gradient descent (or ascent) in CC, but experience has shown that this results in the creation of more hidden units for a given problem. There are two reasons for this: First, backprop learning is much slower, so the experimenter is more likely to stop the training phase while the weights are still some distance from their optimal values. Second, the correlation measure used in candidate training often has small local maxima in which units can get stuck. Gradient ascent without momentum is particularly likely to get stuck in such traps; quickprop, like gradient ascent with high momentum, tends to jump over these small maxima.

Quickprop update does have some disadvantages: First, quickprop assumes that the same stored training set is presented repeatedly. Weights are updated once after each presentation of the entire set. In some situations, the training data arrive in a continuous stream and it is impractical to store them and train repeatedly on the same set.

Second, quickprop requires the storage of four values for every weight: the weight itself, the slope ($\partial E/\partial w$) accumulated for the current training epoch, the previous Δw, and the previous slope. "Batch" gradient descent requires the only the first two of these stored values, or the first three if momentum is used. "Online" or "continuous" gradient descent requires only storage of the weight itself. If the goal is to use the minimum number of memory bits during training, a simple gradient descent algorithm may be superior to quickprop, even if it results in a larger network or requires higher precision in the weights.

Finally, the quickprop update formula 7 requires more computation than the simple gradient descent formulas. In particular, quickprop requires a divide operation. The added cost is normally insignificant, since it occurs in the outer loop at the end of a training epoch, but it may be a problem in special-purpose chips designed with only addition and multiplication instructions, but with no facilities for fast division.

An obvious question is how limited precision affects CC learning if we replace the quickprop weight-updates with simple gradient descent (no momentum). The results shown are for 6 bits of precision in weights, sigmoids, and weight-updates. Probabilistic rounding and variable gain were used.

The use of simple gradient descent increases the number of hidden units by a factor of 1.7 in the parity and sonar problems. This increase may be an acceptable tradeoff on some hardware and for some problems. However, no solutions were found under these conditions for the more difficult two-spirals problem. A more extensive exploration of this issue would be interesting, but is beyond the scope of this paper.

A related question is whether the techniques of weight normalization and probabilistic update can be used

	Quickprop		Simple	
	hidden	epochs	hidden	epochs
6-Parity	4.8	1072	7.6	1313
Sonar	2.8	1511	4.8	2993
Spirals	19.4	4776	fail	fail

Table 9: Quickprop update vs. simple gradient descent in Cascade-Correlation. Weight, training, and sigmoid precision all restricted to 6 bits, with probabilistic rounding.

6-Parity	∞	$\infty^{\pm 1}$	12	10	8	6	4
Cvgt. Trials	9	8	7	8	7	7	8
Avg. Epochs	402.2	330.0	787.1	676.2	594.3	834.3	897.5

Table 10: Training a fixed-topology 6-4-1 shortcut network using quickprop. We show the number of successful trials out of 10 and average learning time for successful trials.

successfully with quickprop on a fixed-topology network, outside of the Cascade-Correlation framework. Recall that in CC, we train only a single layer of weights at a time and never have to propagate error information backward through connections. When approximate, rounded error values are back-propagated and combined, we might expect some new problems to appear.

We investigate this on the 6-parity problem using a network with 6 inputs, one output, a single hidden layer with 4 units, and direct short-cut connections from the network's inputs to its output. This approximates the networks built for this problem by CC, but without the connections from one hidden unit to another. Quickprop learning and variable gain were used for experiments at various levels of precision. Weight, sigmoid, and weight update precision were all restricted at once.

Once again, no learning occurred with less than 12 bits of precision unless probabilistic rounding was employed. Table 10 shows the results of ten trials with probabilistic rounding. About 70% to 80% of the trials converged, regardless of precision, even as we reduced the precision to 4 bits. The learning time (in epochs) is slower for fixed point arithmetic than for floating point, but we see only a slight increase in learning time as precision is reduced from 12 to 4 bits.[2] These results suggest that weight normalization and probabilistic rounding are useful in general, and that they do not depend on peculiarities of the Cascade-Correlation architecture. The interaction of limited-precision errors as they are back-propagated through the network does not appear to cause any unusual problems.

11. Conclusions

These studies suggest the following conclusions:

[2] In terms of computation, these epochs are 2-4 times more costly than CC epochs, which require only a forward pass and which operate on only a part of the final network at any given time.

- When faced with insufficient weight range, CC will not fail abruptly, but instead will add extra units to compensate for the deficit.
- Limited weight range is handled well, and at relatively low cost, by the variable gain technique described in this paper.
- Limited precision in the sigmoid computations does not appear to be a serious problem, even at 4 bits of precision.
- Limited precision in weights, in the weight update computations, or in all areas at once has little effect until some critical limit is reached, at which point there is an abrupt failure of learning. This effect is observed in CC, in quickprop, and (by other investigators) in standard back-propagation.
- In the three problems we studied, this catastrophic failure occurs at 12-14 bits of precision, plus sign. However, this point varies from one problem to another.
- The failure occurs because most of the weight-update steps are effectively zero. This problem is not specific to the CC architecture; it will take the same form in any learning algorithm that depends on gradient descent.
- Probabilistic weight update provides an effective solution for this problem in CC and quickprop networks. With this technique, we observe only a gradual degradation in CC performance as precision is reduced to 6 bits.
- The probabilistic rounding procedure is moderately expensive, since it requires the production of a random number for each weight-update. However, in batch-mode training, these weight updates occur only at the end of each training epoch, and not in the inner loops of the program. A machine designed specifically for neural net simulation might benefit from fast random-number hardware.
- These results suggest that a machine with 32-bit or 24-bit integer arithmetic and storage should be able to run the CC algorithm successfully without probabilistic rounding. A 16-bit word length (with a longer accumulator) would suffice for the problems studied, but is uncomfortably close to the minimum required precision. Some problems may fail on such a machine.
- With probabilistic rounding, a machine with an 8-bit word length (but longer accumulators) may suffice for most problems. If the precision is not quite sufficient, CC will add a few extra units rather than failing abruptly.

References

[Baker, 1988] Baker, T. and D. Hammerstrom (1988), "Modifications to Artificial Neural Networks Models for Digital Hardware Implementation", Tech. Report CS/E 88-035, Oregon Graduate Center, Department of Computer Science and Engineering.

[Fahlman, 1988] Fahlman, S. E. (1988), "Faster-Learning Variations on Back-Propagation: An Empirical Study" in D. S. Touretzky et al. (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann.

[Fahlman, 1990] Fahlman, S. E. and C. Lebiere (1990), "The Cascade-Correlation Learning Architecture" in D. S. Touretzky (ed), *Advances in Neural Information Processing Systems 2*, Morgan-Kaufmann.

[Gorman, 1988] Gorman, R. P. and T. J. Sejnowski (1988), "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets" in *Neural Networks, Vol. 1*, pp. 75-89.

[Hoehfeld, 1990] Hoehfeld, M. (1990), "Fixed Point Arithmetic in Feedforward Neural Networks", Tech. Report FKS3-108, Siemens AG, Munich.

[Hollis, 1990] Hollis, P. W., J. S. Harper, and J. J. Paulos (1990), "The Effects of Precision Constraints in a Backpropagation Learning Network", in *Neural Computation 2*, pp. 363-373.

[Holt, 1990] Holt, J. L. and J. N. Hwang (1991), "Finite Precision Error Analysis of Neural Network Hardware Implementations", Tech Report FT-10 University of Washington, Seattle, WA 98195.

[Lang, 1988] Lang, K. J., and M. J. Witbrock (1988), "Learning to Tell Two Spirals Apart", in D. S. Touretzky et al. (eds.), *Proc. of the 1988 Connectionist Models Summer School*, Morgan-Kaufmann.

[Shoemaker, 1990] Shoemaker, P. A., M. J. Carlin, and R.L. Shimabukuro (1990), "Back-Propagation Learning with Coarse Quantization of Weight Updates", in *Proc. of IJCNN-90*, Washington D. C.