An Empirical Study of Learning Speed

in Back-Propagation Networks

Scott E. Fahlman

September 1988

CMU-CS-88-162

Abstract

Most connectionist or "neural network" learning systems use some form of the back-propagation algorithm. However, back-propagation learning is too slow for many applications, and it scales up poorly as tasks become larger and more complex. The factors governing learning speed are poorly understood. I have begun a systematic, empirical study of learning speed in backprop-like algorithms, measured against a variety of benchmark problems. The goal is twofold: to develop faster learning algorithms and to contribute to the development of a methodology that will be of value in future studies of this kind.

This paper is a progress report describing the results obtained during the first six months of this study. To date I have looked only at a limited set of benchmark problems, but the results on these are encouraging: I have developed a new learning algorithm that is faster than standard backprop by an order of magnitude or more and that appears to scale up very well as the problem size increases.

This research was sponsored in part by the National Science Foundation under Contract Number EET-8716324 and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under Contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OH 45433-6543.

1. Introduction

Note: In this paper I will not attempt to review the basic ideas of connectionism or back-propagation learning. See [3] for a brief overview of this area and [10], chapters 1 - 8, for a detailed treatment. When I refer to "standard back-propagation" in this paper, I mean the back-propagation algorithm with momentum, as described in [9].

The greatest single obstacle to the widespread use of connectionist learning networks in real-world applications is the slow speed at which the current algorithms learn. At present, the fastest learning algorithm for most purposes is the algorithm that is generally known as "back-propagation" or "backprop" [6, 7, 9, 18]. The back-propagation learning algorithm runs faster than earlier learning methods, but it is still much slower than we would like. Even on relatively simple problems, standard back-propagation often requires the complete set of training examples to be presented hundreds or thousands of times. This means that we are limited to investigating rather small networks with only a few thousand trainable weights. Some problems of real-world importance can be tackled using networks of this size, but most of the tasks for which connectionist technology might be appropriate are much too large and complex to be handled by our current learning-network technology.

One solution is to run our network simulations on faster computers or to implement the network elements directly in VLSI chips. A number of groups are working on faster implementations, including a group at CMU that is using the 10-processor Warp machine [13]. This work is important, but even if we had a network implemented directly in hardware our slow learning algorithms would still limit the range of problems we could attack. Advances in learning algorithms and in implementation technology are complementary. If we can combine hardware that runs several orders of magnitude faster and learning algorithms that scale up well to very large networks, we will be in a position to tackle a much larger universe of possible applications.

Since January of 1988 I have been conducting an empirical study of learning speed in simulated networks. I have studied the standard backprop algorithm and a number of variations on standard back-propagation, applying these to a set of moderate-sized benchmark problems. Many of the variations that I have investigated were first proposed by other researchers, but until now there have been no systematic studies to compare these methods, individually and in various combinations, against a standard set of learning problems. Only through such systematic studies can we hope to understand which methods work best in which situations.

This paper is a report on the results obtained in the first six months of this study. Perhaps the most important result is the identification of a new learning method -- actually a combination of several ideas -- that on a range of encoder/decoder problems is faster than standard back-propagation by an order of magnitude or more. This new method also appears to scale up much better than standard backprop as the size and complexity of the learning task grows.

I must emphasize that this is a progress report. The learning-speed study is far from complete. Until now I have concentrated most of my effort on a single class of benchmarks, namely the encoder/decoder problems. Like any family of benchmarks taken in isolation, encoder/decoder problems have certain peculiarities that may bias the results of the study. Until a more comprehensive set of benchmarks has been run, it would be premature to draw any sweeping conclusions or make any strong claims about the widespread applicability of these techniques.

2. Methodology

2.1. What Makes a Good Benchmark?

At present there is no widely accepted methodology for measuring and comparing the speed of various connectionist learning algorithms. Some researchers have proposed new algorithms based only on a theoretical analysis of the problem. It is sometimes hard to determine how well these theoretical models fit actual practice. Other researchers implement their ideas and run one or two benchmarks to demonstrate the speed of the resulting system. Unfortunately, no two researchers ever seem to choose the same benchmark or, if they do, they use different parameters or adopt different criteria for success. This makes it very hard to determine which algorithm is best for a given application.

The measurement problem is compounded by widespread confusion about the speed of standard back-propagation. Selection of the back-propagation learning parameters is something of a black art, and small differences in these parameters can lead to large differences in learning times. It is not uncommon to see learning times reported in the literature that differ by an order of magnitude or more on the same problem and with essentially the same learning method.

The net effect of all this confusion is that we are faced with a vast, uncharted space of possible learning algorithms in which only a few isolated points have been explored, and even for those points it is hard to compare the claims of the various explorers. What we need now is a careful, systematic effort to fill in the rest of the map. The primary goal of this study is to develop faster learning algorithms for connectionist networks, but I also hope to contribute to the development of a new, more coherent methodology for studies of this kind.

One good way to begin is to select a family of benchmark problems that learning-speed researchers can use in a standard way to evaluate their algorithms. We should try to choose benchmarks that will give us good insight into how various learning algorithms will perform on the real-world tasks we eventually want to tackle.

At present, the benchmark that appears most often in the literature is the exclusive-or problem, often called "XOR" : we are given a network with two input units, one or two hidden units, and one output unit, and the problem is to train the weights in this network so that the output unit will turn on if one or the other of the inputs is on, but not both. Some researchers, notably Tesauro and Janssens [16], generalize this to the N-input parity problem: the output is to be on if an odd number of inputs are on.

The XOR/parity problem looms large in the history and theory of connectionist models (see [11] for an important piece of this history), but if our goal is to develop good learning algorithms for real-world pattern-classification tasks, XOR/parity is the wrong problem to concentrate on. Classification tasks take advantage of a learning network's ability to generalize from the input patterns it has seen during training to nearby patterns in the space of possible inputs. Such networks must occasionally make sharp distinctions between very similar input patterns, but this is the exception rather than the rule. XOR and parity, on the other hand, have exactly the opposite character: generalization is actually punished, since the nearest neighbors of an input pattern must produce the opposite answer from the pattern itself.

Other popular benchmark problems, such as the "Penzias" or cluster-counting task have some of this same anti-generalizing quality. Here again, the change of any one bit will almost always make a big change in the answer. As part of a suite of benchmarks, such tasks are valuable, but if used in isolation they may encourage us to develop learning algorithms that do not generalize well.

In my view, we would do better to concentrate on what I will call "noisy associative memory" benchmarks: we select a set of input patterns, perhaps binary strings chosen at random, and expand each of these into a cluster of similar inputs by adding random noise or flipping some of the input bits; we then train the network to produce a different output pattern for each of these input clusters. We might occasionally ask the network to map several different clusters into the same output. Some of the noise-modified input patterns are not used during training; these can be used later to determine whether the network is capturing the "central idea" of the cluster or is just memorizing the specific input patterns it has seen. I believe that performance of a learning algorithm on this type of problem will correlate very well with performance on real-world pattern classification tasks.

This family of benchmarks can be scaled in various ways. We can vary the number of pattern-clusters, the number of input and output units, and the amount of noise added to the inputs. We can treat this as a digital problem, in which the inputs are either 0 or 1, or we can use analog input patterns. We can vary the number of hidden units and the number of layers. Obviously, it will take some time before we have accumulated a solid set of baseline results against which new algorithms can be measured.

2.2. The Encoder/Decoder Task

In the coming months I intend to concentrate on "noisy associative memory" benchmarks, plus some benchmarks adapted from real-world applications in domains such as speech understanding and road following. However, for the early stages of the study it seemed wise to stick with encoder/decoder problems. This family of problems has been popular in the connectionist community for some years, so we have a base of shared experience in applying older learning algorithms to them.

The encoder/decoder problems, often called simply "encoders", will be familiar to most readers of this report. When I speak of an "N-M-N encoder", I mean a network with three layers of units and two layers of modifiable weights. There are N units in the input layer, M hidden units, and N units in the output layer. There is a connection from every input unit to every hidden unit and a connection from every hidden unit to every output unit. In addition, each unit has a modifiable threshold. There are no direct connections from the input units to the output units. Normally, M is smaller than than N, so the network has a bottleneck through which information must flow.

This network is presented with N distinct input patterns, each of which has only one of the input units turned on (set to 1.0); the other input bits are turned off (set to 0.0). The task is to duplicate the input pattern in the output units. Since all information must flow through the hidden units, the network must develop a unique encoding for each of the N patterns in the M hidden units and a set of connection weights that, working together, perform the encoding and decoding operations.

If an encoder network has $M = \log_2 N$, I will speak of it as a "tight" encoder. For example, the 8-3-8 and 16-4-16 encoders are tight in this sense. If the hidden units assume only two values, 1 and 0, then the network must assign every input pattern to one of the N possible binary codes in the hidden layer, wasting none of the codes. In practice, a backprop network will often use three or more distinct analog values in some of the hidden units, so "tight" encoder networks are not really forced to search for optimal binary encodings. It is even possible to learn to perform the encoder/decoder task in an "ultra-tight" network such as 8-2-8, though this takes much longer than learning the same task in an 8-3-8 network.

In a real-world pattern classification task, we will usually give the network enough hidden units to perform the task easily. We do not want to provide too many hidden units -- that allows the network to memorize the training examples rather than extracting the general features that will allow it to handle cases it has not seen during training -- but neither do we want to force the network to spend a lot of extra time trying to find an optimal representation. This suggests that relatively "loose" encoder problems, such as 10-5-10 or 64-8-64, might be more realistic and informative benchmarks than tight or ultra-tight encoders. Much of the work reported here was done on the 10-5-10 encoder, which is not too tight and not too small. This made it possible to run a large number of experiments with this same problem, including some using very slow learning algorithms.

The standard encoder problem has one unusual feature that is not seen in typical pattern classification tasks: in each learning example, only one of the connections on the input side carries a non-zero activation for any given training pattern. In the standard back-propagation algorithm, only this one input-side weight is modified by errors due to that pattern. This separation of effects makes the standard encoder/decoder somewhat easier than the typical pattern-classification task, and therefore perhaps unrealistic as a benchmark. In an effort to understand what kind of bias this may be introducing, I have also looked at *complement encoder* problems, in which the input and output patterns are all ones except for a single zero in some position. As we will see, complement encoders require more learning time than standard encoders, but the difference can be minimized if the right learning techniques are used.

In all of the encoder and complement encoder problems, learning times are reported in *epochs*. An epoch is one presentation of the entire set of N training patterns. All of the algorithms that I have studied to date perform a forward pass and a backward pass on each pattern, collecting error data; the weights are updated at the end of the epoch, after the entire set of N patterns has been seen.

2.3. When is the Learning Complete?

One source of confusion in the learning-speed studies done to date is that each researcher has chosen a different criterion for "successful" learning of a task. In complex pattern-classification tasks, this is a hard problem for several reasons. First, if the inputs are noisy, it may be impossible to perform a perfect classification. Second, if the outputs are analog in nature, the accuracy required will depend on the demands of the specific task. Finally, in many tasks one can manipulate the learning parameters to trade off speed and accuracy against generalization; again, any reasonable criterion for success will depend on the demands of the particular problem. I believe that researchers in this field need to discuss these issues and arrive at some consensus about how various benchmark problems should be run and evaluated.

On something like an encoder/decoder problem there should be little difficulty in agreeing upon criteria for success, but no such agreement exists at present. Some researchers accept any output over 0.50000 as a one and any output below that threshold as a zero. However, if we were to apply this criterion to real analog hardware devices, the slightest bit of additional noise could drive some bits to the other side of the threshold. Other researchers require that each output be very close to the specified target value. For networks performing a task with binary outputs, this seems unnecessarily strict; some learning algorithms may produce useful outputs quickly, but take much longer to adjust the output values to within the specified tolerances. Still other researchers declare success when the sum of the squared error for all the outputs falls below some fixed value. This seems an odd choice, since in a binary application we want each of the individual outputs to be correct; we don't want to trade more error on one output against less error on another.

Another possibility is to consider a network to have learned the problem when, for each input pattern, the output of the "correct" unit is larger than any other output. This criterion is sometimes met much earlier in the training than any criterion based upon a fixed threshold. However, if we were to use this criterion for training an actual hardware network, we would need some additional circuitry to select the largest output; it seems better to let the learning network itself do this job, if possible. In addition, this criterion is not applicable in problems with multiple one-bits in the output pattern.

I suggest that for problems with binary outputs, we adopt a "threshold and margin" criterion similar to that used by digital logic designers: if the total range of the output units is 0.0 to 1.0, any value below 0.4 is considered to be a zero and any value above 0.6 is considered to be a one; values between 0.4 and 0.6 are considered to be "marginal", and are not counted as correct during training. If the output range is different, we scale these two thresholds appropriately. By creating a "no man's land" between the two classes of outputs, we produce a network that can tolerate a small amount of noise in the outputs.

All of the examples in this paper use this 0.4 - 0.6 criterion to measure success. Training continues until, for an entire epoch, we observe that every output is correct; then we stop and declare learning to have been successful.

Since back-propagation networks are deterministic, we will get the same successful results in all future trials as long as we do not change the weights.

2.4. How Should We Report Learning Times?

The *epoch*, defined as a single presentation of each of the I/O patterns in the training set, is a convenient unit by which to measure learning time for the benchmarks reported here. An alternative is to use the *pattern presentation* as the basic unit of learning time. This is the presentation of a single I/O pattern, with the associated forward propagation of results and back-propagation of error, and (sometimes) the modification of weights. The presentation is a more natural measure to use in problems that do not have a finite training set, or that do not cycle through the entire training set between weight-updates. As long as it is made clear what units are being reported, it is easy enough to convert from epochs to presentations, so researchers can choose whatever units they like without fear of confusion.

In measuring a new learning algorithm against a particular benchmark problem, it is desirable to run a large number of trials with the weights initialized to different random values in each case. Any single trial may be misleading: the choice of initial weights might have a greater influence on the time required than any difference between two algorithms. Most of the results reported in this paper are averages over 25 or 100 trials; for some very large or very slow problems, I have been forced to use fewer trials.

Reporting of learning times is a simple enough matter when all of the trials succeed. That is the case for all of the encoder examples I have run. In this case, it is useful to report the average learning time over all the trials, the best and worst results, and the standard deviation or some other measure of the variation in results. Unfortunately, in some problems such as XOR, the network will occasionally become stuck in a local minimum from which it cannot escape. These learning trials never converge, so the learning time is infinite. A few other trials may take an anomalously long time; mixing these long trials into an average may give a distorted picture of the data.

How should such results be reported? One option, used by Robert Jacobs [5], is simply to report the failures in one column and the average of the successful trials in another. The problem with this is that it becomes hard to choose between a learning method with fewer failures and one with a better average. In addition, it becomes unclear whether the average has been polluted by a few very long near-failures.

A second option is adopted by Tesauro and Janssens [16]. Instead of averaging the times for N trials in the usual way, they define the "average" training time to be the inverse of the average training rate. The training rate for each trial is defined as the inverse of the time required for that trial. This method gives us a single, well defined number, even if the set of trials includes some trials that are very long or infinite; the learning rate for these trials goes to zero. However, this kind of average emphasizes short trials and de-emphasizes long ones, so results computed this way look much faster than if a conventional average were used. Note also that if two algorithms are equally fast when measured by a conventional average, the training-rate average will rate the more consistent of the two algorithms as slower. Algorithms that take risky, unsound steps to get very short convergence times on a few trials are favored, even if these algorithms do very poorly by other measures.

The option I favor, for lack of a better idea, is to allow the learning program to restart a trial, with new random weights, whenever the network has failed to converge after a certain number of epochs. The time reported for that trial must include the time spent before the restart and after it. We can consider these restarts to be part of the learning algorithm; the restart threshold is just another parameter that the experimenter can adjust for best results. This seems realistic: when faced with a method that usually converges in 20 epochs or less, but that occasionally gets stuck, it seems only natural to give up and start over at some point. The XOR results reported below use this method of reporting.

2.5. Implementation

All of the experiments reported here were run on a back-propagation simulator that I developed specifically for this purpose. The simulator was written in CMU Common Lisp and runs on the IBM RT PC workstation under the Mach operating system and the X10.4 window system. The machines that were used in this study have been provided by IBM as part of a joint research agreement with the Computer Science Department of Carnegie-Mellon University.

The simulator will soon be converted to use the X11 window system via the CLX interface. It should be relatively easy to port this code to any other implementation of Common Lisp, though I have not taken the time to make the code 100% portable.

Because it is coded in Common Lisp, the simulator is very flexible. It is easy to try out several program variations in a few hours. With the displays turned off, the simulator runs the back-propagation algorithm for a 10-5-10 encoder at roughly 3 epochs per second, a processing rate of about 3500 connection-presentations per second. This is fast enough for experimentation on small benchmarks, especially with the new algorithms that learn such tasks in relatively few epochs. For larger problems and real applications, we will need a faster simulator. By hand-coding the inner loops in assembler, it should be possible to speed up my simulator considerably, perhaps by as much as a factor of ten. Beyond that point, we will have to move the inner loops to a faster machine. For comparison, the Warp machine is now running standard back-propagation at a rate of 17 million connection-presentations per second [13], almost 5000 times faster than my simulator, but it is a very difficult machine to program.

My simulator is designed to make it easy for the experimenter to see what is going on inside of the network. A set of windows displays the changing values of the unit outputs and other per-unit statistics. Another set of windows displays the weights, weight changes, and other per-weight statistics. A control panel is provided through which the operator can alter the learning parameters in real time or single-step the processing to see more clearly what is going on. These displays have been of immense value in helping me to understand what problems were developing during the learning procedure and what might be done about them.

3. Experiments and Results

3.1. Tuning of Backprop Learning Parameters

The first task undertaken in this study was to understand how the learning parameters affected learning time in a relatively small benchmark, the 10-5-10 encoder. There are three parameters of interest: ε, the learning rate; α, the momentum factor; and *r*, the range of the random initial weights.

At the start of each learning trial, each of the weights and thresholds in the network is initialized to a random value chosen from the range *-r* to *+r*.

The formula for updating the weights is $\Delta w(t) = -\varepsilon\,\partial E / \partial w(t) + \alpha\,\Delta w(t{-}1)$, where $\partial E / \partial w(t)$ is the error derivative for that weight, accumulated over the whole epoch. Sometimes I refer to this derivative the "slope" of that weight.

Because standard back-propagation learning is rather slow, I did not exhaustively scan the 3-dimensional space defined by *r*, α, and ε. A cursory exploration was done, with only a few trials at each point tested, in order to find the regions that were most promising. This suggested that an *r* value of 1.0, an α value between 0.0 and 0.5, and an ε value somewhere around 1.0 would produce the fastest learning for this problem. Many researchers have reported good results at α values of 0.8 or 0.9, but on this problem I found that such a high α led to very poor results, often requiring a thousand epochs or more.

The promising area was then explored more intensively, with 25 trials at each point tested. Surprisingly, the best result was obtained with α=0.0, or no momentum at all:

Problem	Trials	ε	α	*r*	Max	Min	Average	S. D.
10-5-10	25	1.7	0.0	1.0	265	80	129	46

This notation says that, with 25 trials and the parameters specified, the longest trial required 265 epochs, the shortest required 80 epochs, the average over all the runs was 129 epochs, and the standard deviation was 46. The standard deviations are included only to give the reader a crude idea of the variation in the values; the distribution of learning times seldom looks like a normal distribution, often exhibiting multiple humps, for example.

The same average learning time was obtained for an α of 0.5 and a smaller ε value:

Problem	Trials	ε	α	*r*	Max	Min	Average	S. D.
10-5-10	25	1.1	0.5	1.0	275	85	129	40

If we hold α at 0.0 and vary ε, we see a U-shaped curve, rising to an average learning time of 242 at ε=0.5 and rising more steeply to a value of 241 at ε=1.3. If we increase α up to 0.5 or so, we see that we are actually in a U-shaped valley whose lowest point is always close to 130 epochs. Above α=0.5, the floor of the valley begins to rise steeply.

Varying the *r* parameter by small amounts made very little difference in any of these trials. Changing *r* by a large amount, either up or down, led to greatly increased learning times. A value of 1.0 or 2.0 seemed as good as any other and better than most.

Plaut, Nowlan, and Hinton [12] present some analysis suggesting that it may be beneficial to use different values of ε for different weights in the network. Specifically, they suggest that the ε value used in tuning each weight should be inversely proportional to the fan-in of the unit receiving activation via that weight. I tried this with a variety of parameter values, but for the 10-5-10 network using standard backprop, it consistently performed worse than using a single, constant ε value for all of the weights. As we will see below, this "split epsilon" technique does turn out to be useful on networks such as the 128-7-128, where the variation in fan-in is very large.

The same paper suggests that the parameter values should be varied as the network learns: α should be very small at first, and should be increased after the network has chosen a direction. The best schedule for increasing α is apparently different for each problem. I did not explore this idea extensively since the quickprop algorithm, described below, seems to do roughly the same job, but in a way that adapts itself to the problem, rather than requiring the human operator to do this job.

3.2. Eliminating the "Flat Spot"

Once I was able to display the weights and unit outputs during the course of the learning, one problem with the standard backprop algorithm became very clear: units were being turned off hard during the early stages of the learning, and they were getting stuck in the zero state. The more hidden units there were in the network, the more likely it was that some output units would get stuck.

This problem is due to the "flat spots" where the derivative of the sigmoid function approaches zero. In the standard back-propagation algorithm, as we back-propagate the error through the network, we multiply the error seen by each unit *j* by the derivative of the sigmoid function at o_j, the current output of unit *j*. This derivative is equal to $o_j(1-o_j)$; I call this the *sigmoid-prime function*. Note that the value of the sigmoid-prime function goes to zero as the unit's output approaches 0.0 or to 1.0. Even if such an output value represents the maximum possible error, a unit whose output is close to 0.0 or 1.0 will pass back only a tiny fraction of this error to the incoming weights and to units in earlier layers. Such a unit will theoretically recover, but it may take a very long time; in a machine with roundoff error and the potential for truncating very small values, such units may never recover.

What can we do about this? One possibility, suggested by James McClelland and tested by Michael Franzini [4], is to use an error measure that goes to infinity as the sigmoid-prime function goes to zero. This is mathematically elegant, and it seemed to work fairly well, but it is hard to implement. I chose to explore a simpler solution: alter the sigmoid-prime function so that it does not go to zero for any output value. The first modification I tried worked the best: I simply added a constant 0.1 to the sigmoid prime value before using it to scale the error. Instead of a curve that goes from 0.0 up to 0.25 and back down to 0.0, we now have a curve that goes from 0.1 to 0.35 and back to 0.1.

This modification made a dramatic difference, cutting the learning time almost in half. Once again we see a valley in α-ε space running from α=0.0 up to about α=0.5, and once again the best learning speed is roughly constant as α increases over this range. The two best values obtained were the following:

Problem	Trials	ε	α	r	Max	Min	Average	S. D.
10-5-10	25	1.5	0.0	1.0	115	50	75	14
10-5-10	25	0.9	0.4	1.0	120	50	74	14

I tried other ways of altering the sigmoid-prime function as well. The most radical was simply to replace this function with a constant value of 0.5; in effect, this eliminates the multiplication by the derivative of the sigmoid altogether. The best performance obtained with this variation was the following:

Problem	Trials	ε	α	r	Max	Min	Average	S. D.
10-5-10	25	0.5	0.4	1.0	170	50	89	34

I next tried replacing sigmoid-prime with a function that returned a random number in the range 0.0 to 1.0. This did not do as well, probably because some learning trials were wasted when the random number happened to be very small. The best result obtained with this scheme was the following:

Problem	Trials	ϵ	α	r	Max	Min	Average	S. D.
10-5-10	25	0.4	0.7	1.0	340	75	163	75

Finally, I tried replacing the sigmoid-prime function with the sum of a constant 0.25 and a random value in the range 0.0 to 0.5. This did about as well as the constant alone:

Problem	Trials	ϵ	α	r	Max	Min	Average	S. D.
10-5-10	25	0.5	0.6	1.0	135	50	90	27

In all cases, the same kind of valley in α-ϵ space was observed, and I was always able to find an ϵ value that gave a near-optimal performance with α=0.0.

The primary lesson from these experiments is that it is very useful to eliminate the flat spots by one means or another. In standard backprop, we must carefully choose the learning parameters to avoid the problem of stuck units; with the modified sigmoid-prime function, we can optimize the parameters for best performance overall. A slight modification of the classic sigmoid-prime function did the job best, but replacing this step with a constant or with a constant plus noise only reduces the learning speed by about 20%. This suggests that this general family of learning algorithms is very robust, and will give you decent results however you scale the error, as long as you don't change the sign or eliminate the error signal by letting the sigmoid-prime function go to zero. Of course, these results may not hold for other problems or for multi-layer networks.

3.3. Using a Non-Linear Error Function

As mentioned in the previous section, several researchers have eliminated the flat spots in the sigmoid-prime function, at least for the units in the output layer of the network, by using an error function that grows to infinity as the difference between the desired and observed outputs goes to 1.0 or -1.0. As the error approaches these extreme values, the product of this non-linear error function and sigmoid-prime remains finite, so some error signal gets through and the unit does not get stuck.

David Plaut suggested to me that the non-linear error function might speed up learning even though the problem of stuck units had been handled by other means. The idea was that for small differences between the output and the desired output, the error should behave linearly, but as the difference increased, the error function should grow faster than linearly, heading toward infinity as errors approach their maximum values.

One function that meets these requirements is hyperbolic arctangent of the difference. I tried using that, rather than the difference itself, as the error signal fed into the output units in the network. Since this function was not competing against one going to zero, I did not let it grow arbitrarily large; I cut it off at -.9999999 and +.9999999, and used values of -17.0 and +17.0 for more extreme differences.

This did indeed have a modest beneficial effect. On the 10-5-10 encoder, using backprop with the hyperbolic arctan error function and adding 0.1 to sigmoid-prime, I was able to get the following result, about a 25% improvement:

Problem	Trials	ϵ	α	r	Max	Min	Average	S. D.
10-5-10	25	0.6	0.0	1.0	85	40	58.7	13

I have not tried applying nonlinear error functions to units in interior layers of the network, but plan to investigate this in the near future. Some sort of non-linear error function may be of value in increasing the learning speed of networks with multiple hidden layers.

3.4. The Quickprop Algorithm

Back-propagation and its relatives work by calculating the partial first derivative of the overall error with respect to each weight. Given this information we can do gradient descent in weight space. If we take infinitesimal steps down the gradient, we are guaranteed to reach a local minimum, and it has been empirically determined that for many problems this local minimum will be a global minimum, or at least a "good enough" solution for most purposes.

Of course, if we want to find a solution in the shortest possible time, we do not want to take infinitesimal steps; we want to take the largest steps possible without overshooting the solution. Unfortunately, a set of partial first derivatives collected at a single point tells us very little about how large a step we may safely take in weight space. If we knew something about the higher-order derivatives -- the curvature of the error function -- we could presumably do much better.

Two kinds of approaches to this problem have been tried. The first approach tries to dynamically adjust the learning rate, either globally or separately for each weight, based in some heuristic way on the history of the computation. The momentum term used in standard back-propagation is a form of this strategy; so are the fixed schedules for parameter adjustment that are recommended in [12], though in this case the adjustment is based upon the experience of the programmer rather than that of the network. Franzini [4] has investigated a technique that heuristically adjusts the global ε parameter, increasing it whenever two successive gradient vectors are nearly the same and decreasing it otherwise. Jacobs [5] has conducted an empirical study comparing standard backprop with momentum to a rule that dynamically adjusts a separate learning-rate parameter for each weight. Cater [2] uses a more complex heuristic for adjusting the learning rate. All of these methods improve the overall learning speed to some degree.

The other kind of approach makes explicit use of the second derivative of the error with respect to each weight. Given this information, we can select a new set of weights using Newton's method or some more sophisticated optimization technique. Unfortunately, it requires a very costly global computation to derive the true second derivative, so some approximation is used. Parker [8], Watrous [17], and Becker and LeCun [1] have all been active in this area. Watrous has implemented two such algorithms and tried them on the XOR problem. He claims some improvement over back-propagation, but it does not appear that his methods will scale up well to much larger problems.

I have developed an algorithm that I call "quickprop" that has some connection to both of these traditions. It is a second-order method, based loosely on Newton's method, but in spirit it is more heuristic than formal. Everything proceeds as in standard back-propagation, but for each weight I keep a copy of the $\partial E / \partial w(t-1)$, the error derivative computed during the previous training epoch, along with the difference between the current and previous values of this weight. The $\partial E / \partial w(t)$ value for the current training epoch is also available at weight-update time.

I then make two risky assumptions: first, that the error vs. weight curve for each weight can be approximated by a parabola whose arms open upward; second, that the change in the slope of the error curve, as seen by each weight, is not affected by all the other weights that are changing at the same time. For each weight, independently, we use the previous and current error slopes and the weight-change between the points at which these slopes were measured to determine a parabola; we then jump directly to the minimum point of this parabola. The computation is very simple, and it uses only the information local to the weight being updated:

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1)$$

where $S(t)$ and $S(t-1)$ are the current and previous values of $\partial E / \partial w$. Of course, this new value is only a crude approximation to the optimum value for the weight, but when applied iteratively this method is surprisingly effective. Notice that the old α parameter is gone, though we will need to keep ε (see below).

Using this update formula, if the current slope is somewhat smaller than the previous one, but in the same direction, the weight will change again in the same direction. The step may be large or small, depending on how much the slope was reduced by the previous step. If the current slope is in the opposite direction from the previous one, that means that we have crossed over the minimum and that we are now on the opposite side of the valley. In this case, the next step will place us somewhere between the current and previous positions. The third case occurs when the current slope is in the same direction as the previous slope, but is the same size or larger in magnitude. If we were to blindly follow the formula in this case, we would end up taking an infinite step or actually moving backwards, up the current slope and toward a local maximum.

I have experimented with several ways of handling this third situation. The method that seems to work best is to create a new parameter, which I call μ, the "maximum growth factor". No weight step is allowed to be greater in magnitude than μ times the previous step for that weight; if the step computed by the quickprop formula would be too large, infinite, or uphill on the current slope, we instead use μ times the previous step as the size of the new step. The idea is that if, instead of flattening out, the error curve actually becomes steeper as you move down it, you can afford to accelerate, but within limits. Since there is some "noise" coming from the simultaneous update of other units, we don't want to extrapolate too far from a finite baseline. Experiments show that if μ is too large, the network behaves chaotically and fails to converge. The optimal value of μ depends to some extent upon the type of problem, but a value of 1.75 works well for a wide range of problems.

Since quickprop changes weights based on what happened during the previous weight update, we need some way to bootstrap the process. In addition, we need a way to restart the learning process for a weight that has previously taken a step of size zero but that now is seeing a non-zero-slope because something has changed elsewhere in the network. The obvious move is to use gradient descent, based on the current slope and some learning rate ε, to start the process and to restart the process for any weight that has a previous step size of zero.

It took me several tries to get this "ignition" process working well. Originally I picked a small threshold and switched from the quadratic approximation to gradient descent whenever the previous weight fell below this threshold. This worked fairly well, but I came to suspect that odd things were happening in the vicinity of the threshold, especially for very large encoder problems. I replaced this mechanism with one that always added a gradient descent term to the step computed by the quadratic method. This worked well when a weight was moving down a slope, but it led to oscillation when the weight overshot the minimum and had to come back: the quadratic method would accurately locate the bottom of the parabola, and the gradient descent term would then push the weight past this point.

My current version of quickprop always adds ε times the current slope to the Δw value computed by the quadratic formula, unless the current slope is opposite in sign from the previous slope; in that case, the quadratic term is used alone.

One final refinement is required. For some problems, quickprop will allow some of the weights to grow very large. This leads to floating-point overflow errors in the middle of a training session. I fix this by adding a small weight-decay term to the slope computed for each weight. This keeps the weights within an acceptable range.

Quickprop can suffer from the same "flat spot" problems as standard backprop, so I always run it with the sigmoid-prime function modified by the addition of 0.1, as described in the previous section.

With the normal linear error function, the following result was the best one obtained using quickprop:

Problem	Trials	ε	μ	r	Max	Min	Average	S. D.
10-5-10	100	1.5	1.75	2.0	72	13	22.1	8.9

With the addition of the hyperbolic arctan error function, quickprop did better still:

Problem	Trials	ε	μ	*r*	Max	Min	Average	S. D.
10-5-10	100	0.35	1.75	2.0	21	9	14.01	2.1

This result is better by about a factor of 4 than any time I obtained with a modified but non-quadratic version of backprop, and it is almost an order of magnitude better than the value of 129 I obtained for standard backprop. With quickprop, only the ε parameter seems to require problem-specific tuning, and even ε does not have to be tuned too carefully for reasonably good results.

3.5. Scaling Experiments

The next step was to see how well the combination of quickprop, adding 0.1 to sigmoid-prime, and hyperbolic arctan error would scale up to larger encoder problems. I decided to run a series of "tight" encoders: 4-2-4, 8-3-8, and so on. For the larger problems in the series, the fan-in for the hidden units was much greater than the fan-in to the output units, and it proved beneficial to divide the value of ε by the fan-in of the unit that is receiving activation from the weight being updated. It also proved useful to gradually reduce ε as the problem size increased.

The results obtained for this series were as follows:

Problem	Trials	ε	μ	*r*	Max	Min	Average	S. D.
4-2-4	100	2.0	1.75	2.0	36	8	15.93	6.2
8-3-8	100	2.0	1.75	2.0	42	12	21.99	5.6
16-4-16	100	1.2	1.75	2.0	48	20	28.93	5.4
32-5-32	25	1.0	1.75	2.0	38	26	30.48	3.1
64-6-64	10	0.75	1.75	2.0	37	28	33.90	2.9
128-7-128	5	0.5	1.75	2.0	43	36	40.20	2.8
256-8-256	5	0.3	1.75	2.0	44	40	42.00	1.6

These times are significantly better than any others I have seen for tight encoder problems. The literature of the field gives very few specific timings for such problems, especially for large ones. The best time I have obtained for the 16-4-16 encoder with standard backprop is 794.6 epochs (average time over 10 trials). With the sigmoid-prime function modified to add 0.1, the time goes down to 539.2.

David Plaut, who has run many backprop simulations during his graduate student career at CMU, is able to get times "generally in the low 40's" on the 16-4-16 encoder using backprop with a non-linear error function. However, he accomplishes this by watching the progress of each learning trial on the display and adjusting α by hand as the learning progresses. This method is hard to replicate, and it is unclear how well it scales up. I suspect that an analysis of Plait's real-time adjustments would show that he is doing something very similar to what quickprop does.

Juergen Schmidhuber [14] has investigated this same class of problems up to 64-6-64 using two methods: first, he used standard backprop, but he adjusted the weights after every presentation of a training example rather than after a full epoch; second, he used a learning technique of his own that measures the total error, rather than the first derivative, and tries to converge toward a zero of the error function. On the 16-4-16 encoder, Schmidhuber reports a learning time of 239 epochs for backprop and 146 for his own method; on 64-6-64, he gets 750 for backprop and 220 for his own method.

The most exciting aspect of the learning times in the table above are the way they scale up as the problem size increases. If we take N as the number of patterns to be learned, the learning time measured in epochs is actually growing more slowly than $\log N$. In the past, it was generally believed that for tight encoder problems this time would grow exponentially with the problem size, or at least linearly.

Of course, the measurement of learning time in epochs can be deceiving. The number of training examples in each epoch grows by a factor of N, and the time required to run each forward-backward pass on a serial machine is proportional to the number of connections -- also roughly a factor of N. This means that on a serial machine, using the techniques described here, the actual clock time required grows by a factor somewhere between N^2 and $N^2 \log N$. On a parallel network, the clock time required grows by a factor between N and $N \log N$. If this scaling result holds larger networks and for other kinds of problems, that is good news for the future applicability of connectionist techniques.

In order to get a feeling for how the learning time was affected by the number of units in the single hidden-unit layer, I ran the 8-M-8 problem for different M values. Again, these results are for quickprop, hyperbolic arctan error, 0.1 added to sigmoid-prime, and epsilon divided by the fan-in.

Problem	Trials	ϵ	μ	r	Max	Min	Average	S. D.
8-2-8	25	1.0	1.75	4.0	155	26	102.8	37.7
8-3-8	100	2.0	1.75	2.0	42	12	21.99	5.6
8-4-8	100	3.0	1.75	2.0	23	10	14.88	2.8
8-5-8	100	3.0	1.75	2.0	19	9	12.29	2.0
8-6-8	100	3.0	1.75	2.0	15	7	10.13	1.6
8-8-8	100	3.0	1.75	2.0	12	6	8.79	1.3
8-16-8	100	3.0	1.75	2.0	10	4	6.28	1.0

The most interesting result here is that the learning time goes down monotonically with increasing M, even when M is much greater than N. Some researchers have suggested that, beyond a certain point, it actually makes learning slower if you add more hidden units. This belief probably came about because in standard backprop, the additional hidden units tend to push the output units deeper into the flat spot. Of course, on a serial simulation, the clock time may increase as more units are added because of the extra connections that must be simulated.

3.6. The Complement Encoder Problem

As I mentioned earlier, the standard encoder problem has the peculiar feature that only one of the connections on the input side is active for each of the training patterns. Since the quickprop scheme is based on the assumption that the weight changes are not strongly coupled to one another, we might guess that quickprop looks better on encoder problems than on anything else. To test this, I ran a series of experiments on the 10-5-10 complement encoder problem, in which each of the input and output patterns is a string of one-bits, with only a single zero. If the standard encoder is unusually easy for quickprop, then the complement encoder should be unusually hard.

The 10-5-10 complement encoder problem was run for each of the following learning algorithms: standard backprop, backprop with 0.1 added to the sigmoid-prime function, the same with the hyperbolic arctangent error function, and quickprop with hyperbolic arctan error. In each case 25 trials were run, and a quick search was run to determine the best learning parameters for each method. Epsilon values marked with an asterisk are divided by the fan-in. These results are summarized in the table below; for comparison, the rightmost column shows the time required by each method for the normal encoder problem.

Method	ε	μ or α	*r*	Max	Min	Average	Normal
Standard. BP	0.9	0.1	1.0	656	140	334.4	129.0
BP, Sig-Prime+0.1	1.1	0.0	1.0	462	111	196.2	74.0
BP, Sig-Prime+0.1, Hyper Err	0.15	0.6	1.0	267	73	134.9	58.7
QP, Hyper Err	0.01*	2.5	2.0	103	36	63.6	14.01
QP, Hyper Err, Symmetric	5.0*	1.15	0.5	31	15	20.8	20.8

For each method tried, learning the complement encoder took more than twice as long as the normal encoder. I believe that this is because, in the complement encoder, the information gathered during each trial is unable to affect the one weight to which it is most relevant, so such information can have only an indirect effect. In the quickprop case, it takes over 4 times as long to learn the complement encoder as to learn the normal encoder.

The last row in the table shows the time required using quickprop, hyperbolic arctan error, and units whose activation function is symmetric around zero, ranging from -0.5 to +0.5 rather than from 0.0 to 1.0. The input and output patterns, and the thresholds used to detect successful learning are shifted down by 0.5 as well.

Stornetta and Huberman [15] advocate the use of such symmetric units. The empirical results they report are sketchy, but they claim speedups ranging from 10% to 50%, depending on the problem. It occurred to me that, with symmetric units, the encoder and complement encoder problems would be equivalent, but it was not clear whether this meant that the complement encoder would be learned four times faster or that the regular encoder would be learned four times slower. As the table shows, the result is between the two extremes, but closer to the speed for the normal encoder.

It seems likely that the normal encoder is one of the few problems that is well-suited to the asymmetric activation functions, and that symmetric activation should be used for most problems.

3.7. The Exclusive-Or Problem

I have argued that the XOR problem receives too much attention in learning-speed studies, but since this is the most popular benchmark at present, I felt that I must try XOR using these new methods. In this problem, even with very conservative parameter settings, there were still some trials that got caught in a local minimum and showed no signs of convergence, so I used the restart method described in section 2.4.

Using quickprop with hyperbolic arctan error, epsilon divided by fan-in, and symmetric activation, and with the restart limit set at 40 epochs, I got the following result for XOR with two hidden units:

Problem	Trials	ε	μ	*r*	Max	Min	Average	S. D.
XOR	100	4.0	2.25	1.0	66	10	24.22	16.3

In 100 trials there were 14 restarts. The median learning time was 19 epochs.

The time of 24.22 epochs compares very favorably with other times reported for this problem using backprop and backprop-like algorithms. Jacobs [5] reports a time of 538.9 (plus one failure) for standard backprop and 250.4 (plus two failures) for his "delta-bar-delta" algorithm. Tesauro and Janssens [16] report an time of 95 epochs, using their rate-based averaging function. Watrous [17] reports a learning time of 3495 epochs for a slightly different version of XOR that uses a single hidden unit and some additional connections; his "BFGS" method learns the problem in 36 epochs, but this involves a very expensive non-local computation for each update epoch. Rumelhart, Hinton, and Williams [9] report that Yves Chauvin got learning times around 245 for the XOR problem with two hidden units. Of course, all of these studies used slightly different success criteria, usually stricter than the criterion

I used.

4. Conclusions and Future Work

On the problems tested to date, the learning algorithms described here run much faster than standard backprop, and scale up much better. These results are encouraging, but are not conclusive because we have only tested the new techniques against a very small, and perhaps atypical, set of benchmark problems. The most important task for the immediate future is to apply the new algorithms, perhaps with some additional modifications, to a variety of additional benchmarks and then to some real-world applications. If the speedup and scaling results hold up in these tests, then we will have achieved something of a breakthrough, especially when these algorithms are implemented on the fastest available machines.

The development of these new algorithms was driven not by a theoretical analysis, but by observing problems that occurred in standard back-propagation learning and by attempting to cure these problems one by one. The result is admittedly something of a patchwork. Now that we have seen what can be accomplished, it would be useful to try to develop a theoretical understanding of some of these tricks. For example, it might be possible to develop a better understanding of the expected performance of quickprop for various classes of problems. This sort of understanding should ultimately lead us to more elegant and faster learning algorithms.

Among the issues that will become more important in the future are incremental learning -- adding new knowledge to a network that has already been trained -- and the development of networks that handle recognize and produce time-varying sequences of inputs, rather than individual patterns. It will be interesting to see whether any of these learning techniques are applicable in these more complex domains.

Acknowledgments

I would like to thank Geoff Hinton for sparking my interest in networks of this kind and Dave Touretzky for his persistent efforts to promote an active interchange of ideas within the local connectionist community. David Plaut, Mark Derthick, Barak Pearlmutter, and Kevin Lang offered valuable suggestions and helped to acquaint me with the folklore of back-propagation learning systems.

References

[1] Becker, S. and LeCun, Y.
The Feasibility of Applying Numerical Optimization Techniques to Back-Propagation.
In *Proceedings, 1988 Connectionist Models Summer School*. Morgan-Kaufman, 1988.
(to appear).

[2] Cater, J. P.
Successfully Using Peak Learning Rates of 10 (and greater) in Back-Propagation Networks with the Heuristic Learning Algorithm.
In *Proceedings of the IEEE International Conference on Neural Networks*, pages 645-651. San Diego, CA, 1987.

[3] Fahlman, S. E. and Hinton, G. E.
Connectionist Architectures for Artificial Intelligence.
IEEE Computer 20(1):100-109, January, 1987.

[4] Franzini, M. A.
Speech Recognition with Back Propagation.
In *Proceedings, Ninth Annual Conference of IEEE Engineering in Medicine and Biology Society*. 1987.

[5] Jacobs, R. A.
Increased rates of Convergence Through Learning Rate Adaptation.
Technical Report COINS TR 87-117, University of Massachusetts at Amherst, Dept. of Computer and Information Science, Amherst, MA, 1987.

[6] LeCun, Y.
Une procedure d'apprentissage pour reseau a seauil assymetrique (A learning procedure for asymmetric threshold network).
In *Proceedings of Cognitiva 85*, pages 599-604. Paris, 1985.

[7] Parker, D. B.
Learning-Logic.
Technical Report TR-47, Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science, Cambridge, MA, 1985.

[8] Parker, D. B.
Optimal ALgorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning.
In *Proceedings of the IEEE International Conference on Neural Networks*, pages 593-600. San Diego, CA, 1987.

[9] Rumelhart, D. E., Hinton, G. E., and Williams, R. J.
Learning Internal Representations by Error Propagation.
In Rumelhart, D. E. and McClelland, J. L. (editor), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, chapter 8. MIT Press, Cambridge, MA, and London, England, 1986.

[10] Rumelhart, D. E. and McClelland, J. L.
Parallel Distributed Processing: Explorations in the Microstructure of Cognition.
MIT Press, Cambridge, MA, and London, England, 1986.

[11] Minsky, M. L. and Papert, S.
Perceptrons.
MIT Press, Cambridge, MA, and London, England, 1969.

[12] Plaut, D. C., Nowlan, S. J., and Hinton, G. E.
Experiments on Learning by Back-Propagation.
Technical Report CMU-CS-86-126, Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA, 1986.

[13] Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S, and Kung, H. T.
Neural network simulation at Warp speed: How we got 17 million connections per second.
In *Proceedings of the IEEE International Conference on Neural Networks*. San Diego, CA, 1988.
(to appear).

[14] Schmidhuber, J.
Accelerated Learning in back-Propagation Nets.
Technical Report , Technische Universitaet Muenchen, Institut Fuer Informatik, Munich, Federal Republic of Germany, 1988.

[15] Stornetta, W. S., and Huberman, B. A.
An Improved Three-Layer Back-Propagation Algorithm.
In *Proceedings of the IEEE International Conference on Neural Networks*, pages 637-644. San Diego, CA, 1987.

[16] Tesauro, G. and Janssens, B.
Scaling Relationships in Back-Propagation Learning.
Complex Systems 2 :39-44, 1988.

[17] Watrous, R. L.
Learning Algorithms for Connectionist Networks: Applied Gradient Methods for Non-Linear Optimizaiton.
In *Proceedings of the IEEE International Conference on Neural Networks*, pages 619-627. San Diego, CA, 1987.

[18] Werbos, P. J.
Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.
PhD thesis, Harvard University, 1984.

Table of Contents