Training Multilayered Neural Networks
by Replacing the Least Fit Hidden Neurons

Dr. Donald L. Prados
Electrical Engineering Department
University of New Orleans
New Orleans, LA 70148

ABSTRACT

This paper discusses a new method for training multilayered neural networks that uses genetic-algorithm techniques. Tests show that this method, called GenLearn, is significantly faster than methods that use the generalized delta rule (GDR). GenLearn also has the advantage that, unlike the GDR, it performs a global search of weight space. Since GenLearn performs a global search of weight space, it avoids the common problem of getting stuck in local minima.

In a feedforward, three-layered neural net, the search for an appropriate first-layer weight matrix can be thought of as a search for powerful internal representations of the input data. GenLearn, in searching for the most fit hidden neurons, searches for a globally-optimal internal representation. The hidden neurons that are least fit (those that form the least fit internal representations) do not survive and are replaced. Thus, GenLearn is based on survival of the fittest hidden neuron.

A big advantage of the GenLearn procedure over the GDR in training three-layered neural nets is that, during each iteration of GenLearn, each weight in the first matrix is modified only once; whereas, in the GDR procedure, each weight in the first matrix is modified once for each output-layer neuron. Both procedures modify each weight in the second matrix once per iteration. What makes this such a big advantage is that, although GenLearn often reaches the desired mean square error in about the same number of iterations as the GDR, each iteration takes considerably less time.

Introduction.

The most common neural-network learning algorithm, the generalized delta rule [1] (also known as "backpropagation"), is actual nothing more than a gradient-descent search. A variety of other search techniques have been studied extensively by the artificial intelligence (AI) community for decades. As the field of artificial neural networks matures, neural-network search techniques are being combined with other search techniques such as those that use branch-and-bound [2], simulated annealing [3], and genetic algorithms [4].

This paper discusses a new method for training multilayered neural networks that uses genetic-algorithm techniques. Tests show that this method is significantly faster than methods that use the generalized delta rule (GDR). It also has the advantage that, unlike the GDR, it performs a global search of weight space.

To successfully store a set of binary patterns in a neural net of one or two layers using supervised learning requires each output neuron to produce an output of +1 for some of the input patterns and -1 (or 0) for the rest. This requires the set of input patterns to be linearly separable with respect to each output neuron and, therefore, significantly limits the number and types of functions that the neural network can calculate. For this reason and others, much of the recent research in neural-network learning has focused on multilayered neural networks.

The generalized delta rule, the most popular method of training multilayered neural networks, has two major problems. The first is that it is often very slow to converge. The second is that it can get stuck in local minima and not converge at all. The GDR often gets stuck in local minima because it is actually nothing more than a gradient descent to minimize error from a randomly-chosen starting point in weight space.

A method which searches for globally-optimal weight matrices potentially can out-perform the gradient-descent algorithms such as the GDR. This paper discusses a supervised-learning algorithm for training multilayered neural-networks, called GenLearn. GenLearn uses techniques from the field of genetic algorithms to perform a global search of weight space and, thereby, to avoid the common problem of getting stuck in local minima. GenLearn is based on "survival of the fittest" hidden neuron. In searching for the "most fit" hidden neurons, GenLearn searches for a globally-optimal internal representation of the input data.

A big advantage of the GenLearn procedure over the GDR in training three-layered neural nets is that, during each iteration of GenLearn, each weight in the first matrix is modified only once; whereas, in the GDR procedure, each weight in the first matrix is modified once for each output-layer neuron. What makes this such a big advantage is that, although GenLearn often reaches the desired mean square error in about the same number of iterations as the GDR, each iteration takes considerably less time.

The Delta Rule for Two-layered Neural Nets.

This section discusses the training of neural networks with only an input layer and an output layer using the delta rule. The equations introduced here are modified later for training multilayered neural nets.

It is assumed that one is given a set of binary (+1s and -1s) pattern pairs for the neural net to learn. To successfully store a set of pattern pairs, the neural net must form a mapping from input patterns to output patterns such that, for each input pattern, the neural net produces the desired (or target) output pattern.

The actual output pattern, o, is calculated as

$$\mathbf{o} = F(\mathbf{W}\mathbf{i}), \tag{1}$$

where W is the connection matrix, i is the input pattern, and F() is the activation function, with elements f() given by

$$f(x) = \frac{x}{(1+|x|)}, \tag{2}$$

where x is an element of Wi. The activation function acts separately on each element of the vector obtained by multiplying W by i. |x| is the absolute value of x.

Noticed that -1 < f(x) < +1 for all real numbers x. If the target output pattern t consists only of +1s and -1s, there will always be some error associated with each output neuron. The error associated with output neuron k is $t_k - o_k$. The mean square error can be used to measure how well a pattern pair is stored. A pattern pair can be considered to be successfully stored if each actual output bit has the same sign as the target output bit.

The delta rule is used to iteratively update W until either the mean square error is acceptable, the pattern pairs are all successfully stored, or a limit set upon the number of iterations is reached. The weights in row k of W are changed according to the formula

$$\mathbf{w}_k(n+1) = \mathbf{w}_k(n) + \alpha f'(\mathbf{w}_k \cdot \mathbf{i})[t_k - f(\mathbf{w}_k \cdot \mathbf{i})]\,\mathbf{i}. \tag{3}$$

where n indicates the iteration number and t_k is bit k of the target output pattern t. Note that the derivative of f(x) is

0-7803-0494-2/92 $3.00 ©1992 IEEE

$$f'(x) = \frac{1}{(1+|x|)^2}. \qquad (4)$$

The delta rule changes the weights in such a way as to cause $f(w_k \cdot i)$ to approach t_k. Also, as $f(w_k \cdot i)$ approaches t_k, the magnitudes of the weight changes decrease.

The learning algorithm for the delta rule is a gradient-descent algorithm which causes movement in weight space in the direction of the negative gradient of the energy function

$$E = \frac{\alpha}{2}[\boldsymbol{t} - F(\boldsymbol{W}\boldsymbol{i})]^T[\boldsymbol{t} - F(\boldsymbol{W}\boldsymbol{i})]. \qquad (5)$$

This equation gives the energy of pattern pair (i,t). The energy of a set of pattern pairs is, obviously, the sum of their individual energies.

The Generalized Delta Rule for Multilayered Neural Nets.

Since neural nets of two layers can only store linearly-separable patterns, much of the recent research in neural-network learning has focused on multilayered neural nets of three or more layers.

The delta rule of Equation 3 can be generalized to multilayered neural nets as shown by the PDP group [1] and others. Their generalized delta rule (GDR) for a three-layered neural net is as follows.

The output of the hidden layer is

$$\boldsymbol{h} = F(\boldsymbol{W1}\,\boldsymbol{i}), \qquad (6)$$

where **W1** is the weight matrix connecting the input layer to the hidden layer. The output of the third layer is

$$\boldsymbol{o} = F(\boldsymbol{W2}\,\boldsymbol{h}), \qquad (7)$$

where **W2** is the weight matrix connecting the hidden layer to the output layer.

The weights in **W2** are changed in a manner similar to the way the weights in a two-layered neural net are changed (see Equation 3):

$$\boldsymbol{w2}_k(n+1) = \boldsymbol{w2}_k(n) + \alpha f'(\boldsymbol{w2}_k \cdot \boldsymbol{h})\,[t_k - f(\boldsymbol{w2}_k \cdot \boldsymbol{h})]\,\boldsymbol{h}. \qquad (8)$$

The weights in **W1** are changed according to the equation

$$\boldsymbol{w1}_k(n+1) = \boldsymbol{w1}_k(n) + \alpha f'(\boldsymbol{w1}_k \cdot \boldsymbol{i}) \sum_j [w2_{jk} f'(\boldsymbol{w2}_j \cdot \boldsymbol{h})(t_j - f(\boldsymbol{w2}_j \cdot \boldsymbol{h}))]\,\boldsymbol{i} \qquad (9)$$

where $w2_{jk}$ is the weight connecting hidden neuron k to output neuron j.

Once the number of hidden neurons has been chosen, the GDR can be used to find a weight matrix for each layer in the neural network. As mentioned in [1], however, if all weights start out with equal values, the system can never learn. This is because all hidden units connected directly to the output units will get identical error signals, resulting in the weights being changed by identical amounts. In [1], this was handled by initializing the weights to random values.

There are two major problems with the GDR. First, it is often very slow. Second, it can get stuck in local minima. The choice of initial weights can greatly affect both the speed of convergence and the final weight matrices obtained. With a poor choice of initial weights, the GDR may either not be able to find weight matrices to correctly associate the set of input patterns with the target output patterns or else take an inordinate amount of time to do so. The GDR often gets stuck in local minima because it is actually nothing more than a gradient descent to minimize error from a randomly chosen starting point in weight space.

GenLearn.

Genetic algorithms are an established procedure that can be used to perform a global search. They have been widely used in neural networks and are explained in [5]. Perhaps the most obvious way to use genetic algorithms to train neural networks is to use the weights as strings [4], each string consisting of the entire set of weights in the neural net. The major problem with this method is the prohibitive number of weights in large neural networks. For example, if the number of input neurons is 500, the number of hidden neurons is 500, and the number of output neurons is 500, then strings of length 500,000 will be required. If eight bits are used to store each weight, each of the genetic strings will be 4 million bits long.

To avoid such long genetic strings, a method has been developed in which the length of the strings is merely the number of training patterns. Each string is associated with a single hidden neuron. To understand this method, called "GenLearn," one must first understand what a hidden neuron does.

Given a multilayered neural network that has already been trained, each hidden neuron will output a particular value for each input pattern. Thus, each hidden neuron has associated with it a **hidden-neuron pattern**, the length of which is the number of input patterns. The hidden-neuron pattern forms the genetic string associated with the hidden neuron. An important advantage of this technique is that the strings can be binary, which simplifies the job of the genetic algorithm since genetic algorithms perform best on binary strings.

For example, suppose one wishes to train a neural net to perform the exclusive-or (XOR) operation, using two input neurons, two hidden neurons, and one output neuron. One of many solutions is to have one hidden neuron perform the OR operation and the other hidden neuron perform the NAND operation. The XOR can then be obtained by summing the outputs of the two hidden neurons and subtracting 1:

i_1	i_2	OR	NAND	XOR
-1	-1	-1	+1	-1
-1	+1	+1	+1	+1
+1	-1	+1	+1	+1
+1	+1	+1	-1	-1

In this case, the hidden neuron that performs the OR operation has a hidden-neuron pattern (-1 +1 +1 +1), and the hidden neuron that performs the NAND operation has the hidden-neuron pattern (+1 +1 +1 -1). Any good supervised learning algorithm can very quickly train the first weight matrix to perform the OR and NAND operations. A good weight vector for the final output neuron is simply (+1, +1, -1), where the +1s are the weights associated with the two hidden neurons and the -1 is a weight associated with an input that is constantly held at +1. (In all our simulations, we append all patterns by +1 and have a variable weight associated with this constant +1 input). The first weight matrix simply stores the four pattern pairs:

(-1, -1, +1) -> (-1, +1)
(-1, +1, +1) -> (+1, +1)
(+1, -1, +1) -> (+1, +1)
(+1, +1, +1) -> (+1, -1)

The second weight matrix stores the pattern pairs:

(-1, +1, +1) -> -1
(+1, +1, +1) -> +1
(+1, +1, +1) -> +1
(+1, -1, +1) -> -1

In a feedforward, multilayered neural net, the search for an appropriate first-layer weight matrix can be thought of as a search for powerful **internal representations** of the input data. The power of the hidden neurons is due to their formation of these internal representations. GenLearn, in searching for the most fit hidden neurons, searches for an optimal internal representation. Moreover, the hidden neurons that are least fit (those that form the least useful internal representations) do not survive and are replaced. Thus, GenLearn is base on survival of the fittest hidden neurons.

The GenLearn Procedure.

GenLearn basically consists of the following steps:

1. Initialize the weights with random values and generate a set of random hidden-neuron patterns.

2. Run several iterations of a chosen learning rule to attempt to produce the target output patterns from the input patterns.

3. Check termination conditions (for example, mean square error and limit on number of iterations) and either continue to Step 4 or stop.

4. Measure the fitnesses of the hidden neurons.

5. Replace the least fit hidden neurons, and go to Step 2.

The first step in the GenLearn procedure is to generate a random value for each weight and to generate a random hidden-neuron pattern for each hidden neuron. These random hidden-neuron patterns give the initial target outputs for the hidden neurons.

Next, a learning rule is applied to attempt to produce the target hidden-neuron patterns from the input patterns. For example, the delta rule of Equation 3 can be used as follows:

$$\mathbf{w1}_k(n+1) = \mathbf{w1}_k(n) + \alpha f'(\mathbf{w1}_k \cdot \mathbf{i})[h_k - f(\mathbf{w1}_k \cdot \mathbf{i})]\mathbf{i}, \qquad (10)$$

where h_k is the target output for hidden neuron k. After one application of the learning rule, the actual outputs of the hidden neurons are calculated using Equation 6. The actual outputs of the hidden neurons do not have to be the same as the target outputs. Next, a learning rule is applied to W2 to attempt to produce the target output-layer patterns from the *actual* outputs of the hidden neurons. This learning rule can be the same as that used by the GDR, given in Equation 8.

A big advantage of the GenLearn procedure over the GDR in training three-layered neural nets is that, during each iteration of GenLearn, each weight in the first matrix is modified only once; whereas, in the GDR procedure, each weight in the first matrix is modified once for each output-layer neuron. Both procedures modify each weight in the second matrix once per iteration. What makes this such a big advantage is that, although GenLearn often reaches the desired mean square error in about the same number of iterations as the GDR, each iteration takes considerably less time.

A decision has to be made as to how many iterations should take place before replacing the least fit hidden neurons. Experiments have shown that a dozen or so iterations per generation works well for small neural nets. Before replacing the least fit hidden neurons, the mean square error can be calculated to determine if the neural net has already learned the patterns to the accuracy required or if the maximum number of iterations has been reached. The mean square error is calculated as

$$MSE = \frac{1}{mX}\sum_{s=1}^{m}\sum_{x=1}^{X}\left[t_x^s - f(\mathbf{w2}_x \cdot \mathbf{h}^s)\right]^2, \qquad (11)$$

where $\mathbf{h}^s$ is the actual hidden-layer output for input pattern $\mathbf{i}^s$, m is the number of input patterns, X is the number of output neurons, and t_x^s is the target output of output neuron x for input pattern s.

Several methods of calculating the fitness of a hidden neuron work well. Perhaps the simplest is the following:

$$fit_k = \sum_{s=1}^{m}\sum_{x=1}^{X} t_x^s w2_{xk} h_k^s, \qquad (12)$$

where h_k^s is the actual output of hidden neuron k for input pattern $\mathbf{i}^s$. Note that the fitness of a hidden neuron depends not on the string associated with the hidden neuron but on the actual outputs of the hidden neuron as calculated using Equation 6. The fitness equation is base on the assumption that the error is $t_x - f(\mathbf{w2}_x \cdot \mathbf{h})$. If the target output t_x is +1, then one wants $f(\mathbf{w2}_x \cdot \mathbf{h})$ to be close to +1. Similarly, if t_x is -1, one wants $f(\mathbf{w2}_x \cdot \mathbf{h})$ to be close to -1. Thus, the larger the product of t_x and $\mathbf{w2}_x \cdot \mathbf{h}$ is, the closer the actual output is to the desired output. An indication of the fitness of hidden neuron h_k is its contribution to making $t_x \mathbf{w2}_x \cdot \mathbf{h}$ large.

GenLearn replaces the least fit hidden neurons either with new (random) hidden neurons or with hidden neurons obtained by mating and mutating fit hidden neurons. This process of simulated evolution of hidden neurons was first suggested by Selfridge [6] in 1958. Selfridge used the term **subdemon** to refer to a hidden neuron in his Pandemonium model. These subdemons are generated and eliminated through a natural-selection process. In the words of Selfridge:

> First we eliminate those subdemons with low worths. Next we generate new subdemons by mutating the survivors and reweighting the assembly. ... The scheme sketched is really a natural selection on the processing demons. If they serve a useful function they survive, and perhaps are even the source for other subdemons who are themselves judged on their merits.

Mating two hidden neurons involves combining the hidden-neuron patterns associated with the two hidden neurons to obtain a new hidden-neuron pattern. One method of combining two patterns is to first randomly choose a location between the first bit and the last bit and then combine the first part of one pattern with the last part of the other pattern. For example, suppose the hidden-neuron patterns are (-1, -1, +1, +1) and (+1, +1, +1, -1). First choose a random number greater or equal to 1 but less than 4. If this number is 1, for instance, the resulting pattern will be (-1, +1, +1, -1). This pattern is obtained by combining the first bit of the first pattern with the last three bits of the second pattern. If the number is 2 or 3, the resulting pattern will be (-1, -1, +1, -1).

After mating fit hidden neurons to replace the least fit hidden neurons, the iterative procedure continues with the new generation of hidden neurons.

Results.

Tables 1, 2, and 3 show results of tests in which GenLearn is compared to the GDR procedure. Each test consisted of the following:

Figure 1. GenLearn vs. GDR, H=18
N=20, X=20

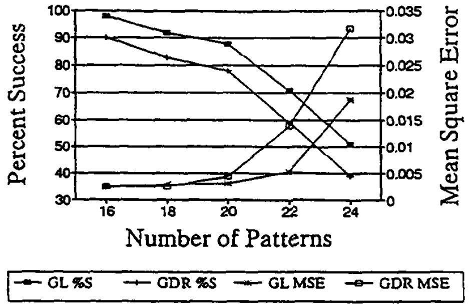


Figure 2. GenLearn vs. GDR, H=20
N=20, X=20

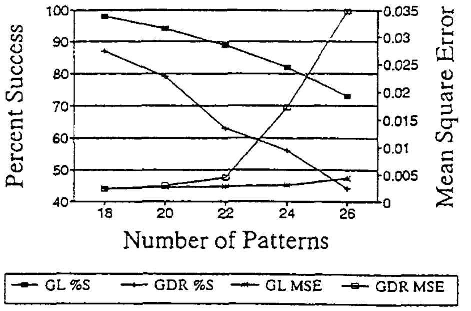
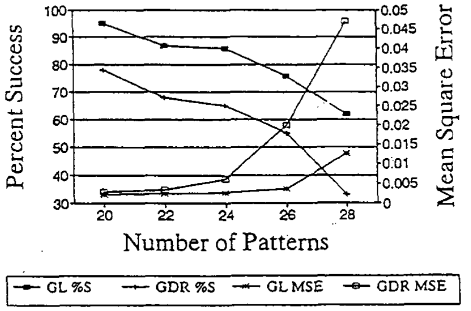


Figure 3. GenLearn vs. GDR, H=22
N=20, X=20



1. First a set of random input and output patterns was generated, each bit of each pattern having equal probabilities of being +1 or -1.

2. The two weight matrices were initialized with random values, each weight having equal probabilities of being +1 or -1.

3. For GenLearn's use, a set of random hidden-neuron patterns was generated, one for each hidden neuron.

4. No more than 1000 iterations of GenLearn were run to attempt to produce the target output patterns from the input patterns. An iteration consisted of applying Equations 10 and 8. Equation 10 was applied once to each row of the first weight matrix, and Equation 8 was applied once to each row of the second weight matrix.

After every 20 iterations, the mean-square error was checked using Equation 11. If the mean square error was below 0.01, learning was terminated; otherwise, the two least-fit hidden-neuron patterns were replaced with two new, randomly-generated hidden-neuron patterns. The fitnesses were then calculated using Equation 12.

5. After learning was terminated, checks were made to see if the set of pattern pairs was successfully stored by GenLearn and to calculate the final mean square error. The set of pattern pairs was considered to be successfully stored if, for each pattern pair, each actual output bit had the same sign as the target output bit.

6. The two weight matrices were again initialized with random values, each weight having equal probabilities of being +1 or -1.

7. No more than 200 iterations of the GDR were run to attempt to produce the target output patterns from the input patterns. An iteration consisted of applying Equations 9 and 8. As in step 4, Equation 8 was applied once to each row of the second weight matrix; however, Equation 9 was applied X times to each row of the first weight matrix where X equals the number of output neurons.

After every 20 iterations, the mean-square error was checked; and, if it was below 0.01, learning was terminated.

8. After learning was terminated, checks were made to see if the set of pattern pairs was correctly stored and to calculate the final mean square error.

The number of input neurons, N, and the number of output neurons, X, were set to 20 for all four tests. The number of hidden neurons, H, was set to 18 for the first test (Figure 1), 20 for the second test (Figure 2), and 22 for the third test (Figure 3). For each test the number of pattern pairs, m, was varied from H-2 to H+4, by increments of two. For each value of m, 200 trials were run, each trial using a new, random set of pattern pairs. The Figures show the percentage of trials in which all pattern pairs were successfully stored (Percent Success) along with how well the pattern pairs were stored (average Mean Square Error).

Discussion

The data in the three graphs was obtained from a single computer run that took approximately 27 hours. GenLearn took a total of about 3 hours and 20 minutes, and the GDR took a total of about 23 hours and 40 minutes. Not only was GenLearn significantly faster, but it also tended to produce much smaller mean square errors and was much more likely to successfully store the patterns. The large difference in speed is despite the fact that GenLearn was allowed to run a maximum of 1000 iterations per trial and the GDR was only allowed a maximum of 200 iterations per trial. When a similar test was run in which both algorithms were allowed the same number of iterations per trial, GenLearn was 20 times faster and had a similar average MSE. This difference is largely due to the fact that GenLearn modifies each weight in the first weight matrix once per iteration whereas the GDR modifies each weight in the first weight matrix X times per iteration, where X is the number of output neurons (set to 20 in these tests).

Notice that the average MSE of both GenLearn and the GDR begins to increase when the number of pattern pairs, m, is increased above the number of hidden neurons, H. For m = H+2, the average MSE is only slightly greater than for m = H; but, as m increases above H+2, the performance rapidly deteriorates. Also, notice the strong relationship between percent success and MSE. As the percentage of unsuccessful trials increases, the average MSE rapidly increases. Finally, notice that as H increases, the performance increases significantly. For example, compare the MSE at m = 24 for each of the three graphs.

Conclusion.

This paper presents a new method for training multilayered neural networks that uses genetic-algorithm techniques. Tests show that this method is significantly faster than the generalized delta rule (GDR), producing a lower mean square error in a shorter period of time. It also has the advantage that, unlike the GDR, it performs a global search of weight space.

References.

[1] D. Rumelhart, J. McClelland, and The PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press, 1986.

[2] J. Sklansky, "Adaptive Piecewise Linear Classifiers and Neural Networks," *Tulane-LSU-UNO Joint Colloquium Series on Neural Networks*, Tulane University, March, 1991.

[3] D. Ackley, G. Hinton, and T. Sejnowski, "A Learning Algorithm for Boltzmann Machines," *Cognitive Science*, vol. 9, pp. 147-169, 1985.

[4] D. Whitley, T. Starkweather, and C. Bogart, "Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity." To appear in: *Parallel Computing*.

[5] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[6] O. Selfridge, "Pandemonium: A Paradigm for Learning," *Mechanisation of Thought Processes: Proc. of a Symposium Held at the National Physics Laboratory*, London: HMSO, pp. 513-526, 1958.