Evolutionary Algorithms for Integer Weight Neural Network Training

V.P. Plagianakos, D.G. Sotiropoulos, and M.N. Vrahatis
University of Patras, Department of Mathematics,
Division of Computational Mathematics & Informatics,
GR-265 00, Patras, Greece.
e-mail: vpp|dgs|vrahatis@math.upatras.gr.

Abstract

In this work differential evolution strategies are applied in neural networks with integer weights training. These strategies have been introduced by Storn and Price [Journal of Global Optimization, 11, pp. 341–359, 1997]. Integer weight neural networks are better suited for hardware implementation as compared with their real weight analogous. Our intention is to give a broad picture of the behaviour of this class of evolution algorithms in this difficult task. Simulation results show that this is a promising approach.

1 Introduction

An artificial Feedforward Neural Network (FNN) consists of many interconnected identical simple processing units, called neurons. Each neuron calculates the dot product of the incoming signals with its weights, adds the bias to the resultant, and passes the calculated sum through the activation function. In a multilayer feedforward network the neurons are organized into layers with no feedback connections.

FNNs can be simulated in software, but in order to be utilized in real life applications, where fast speed of execution is required, hardware implementation is needed. The natural implementation of an FNN – because of its modularity – is a parallel one. The problem is that the conventional multilayer FNNs, which have continuous weights is expensive to implement in digital hardware. Another major implementation obstacle is the weight storage. FNNs having integer weights and biases are easier and less expensive to implement in electronics as well as in optics and the storage of the integer weights is much easier to achieved.

A typical FNN consisting of L layers, where the first layer denotes the input, the last one, L, is the output, and the intermediate layers are the hidden layers. It is assumed that the (l-1) layer has N_{l-1} neurons. Neurons operate according to the following equations

$$net_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1} + \theta_j^l, \quad y_j^l = \sigma^l\Big(net_j^l\Big),$$

where $w_{ij}^{l-1,l}$ is the connection weight from the ith neuron at the $(l-1)$ layer to the j-th neuron at the lth layer, y_i^l is the output of the ith neuron belonging to the lth layer, θ_j^l denotes the bias of the j-th neuron at the lth layer, and σ is a nonlinear activation function. The weights in the FNN can be expressed in vector notation. Let the weight vector have the form: $w = (w_1, w_2, \ldots, w_N)$. The weight vector in general defines a point in the N–dimensional real Euclidean space $\mathbb{R}^N$, where N denotes the total number of weights and biases in the network. Throughout this paper w is considered to be the vector of the integer weights and biases.

From the optimization point of view, supervised training of an FNN is equivalent to minimizing a global error function which is a multivariate function that depends on the weights in the network. The square error over the set of input–desired output patterns with respect to every weight, is usually taken as the function to be minimized. Specifically, the error function for an input pattern t is defined as follows:

$$e_j(t) = y_j^L(t) - d_j(t), \qquad j = 1, 2, \ldots, N_L,$$

where $d_j(t)$ is the desired response of an output neuron at the input pattern t. For a fixed, finite set of input-desired output patterns, the square error over the training set which contains T representative pairs is :

$$E(w) = \sum_{t=1}^{T} E_t(w) = \sum_{t=1}^{T} \sum_{j=1}^{N_L} e_j^2(t),$$

where $E_t(w)$ is the sum of the squares of errors associated with pattern t. Minimization of E is attempted by using a training algorithm to update the weights. The updated weight vector describes a new direction in which the weight vector will move in order to reduce the network training error. Efficient training algorithms have been proposed for trial and error based training, but it is difficult to use them when training with discrete weights [3].

In this work a differential evolution approach, as explained in Section 2, has been utilized to train a neural network with integer weights, suitable for hardware implementation. A brief overview of the most used differential evolution strategies is also presented. Experiments and computer simulation results are presented in Section 3. The final section contains concluding remarks and a short discussion for future work.

2 Differential Evolution Training

In a recent work, Storn and Price [5] have presented a novel minimization method, called Differential Evolution (DE), which has been designed to handle non-differentiable, nonlinear, and multimodal objective functions. To fulfill this requirement, DE has been designed as a stochastic parallel direct search method which utilizes concepts borrowed from the broad class of evolutionary algorithms, but requires few easily chosen control parameters. Experimental results have shown than DE has good convergence properties and outperforms other evolutionary algorithms.

In order to apply DE in neural network training with integer weights, we start with a specific number (NP) of N-dimensional integer weight vectors, as an initial weight population, and evolve them over time. NP is fixed throughout the training process. The weight population is initialized with random integers from the interval $[-\Delta, \Delta]$ following a uniform probability distribution. At each iteration, called *generation*, new weight vectors are generated by the combination of weight vectors randomly chosen from the population and the outcome is rounded to the nearest integer. This operation is called *mutation*. The outcoming integer weight vectors are then mixed with another predetermined integer weight vector – the *target* weight vector – and this operation is called *crossover*. This operation yields the so-called *trial* weight vector, which is an integer vector. The trial vector is accepted for the next generation if and only if it reduces the value of the error function E. This last operation is called *selection*.

We now briefly review the two basic DE operators used for integer weight FNN training. The first DE operator we consider is mutation. Specifically, for each weight vector w_g^i, $i = 1, \ldots, NP$, where g denotes the current generation, a new vector v_{g+1}^i (mutant vector) is generated according

to one of the following relations:

$$v_{g+1}^i = w_g^{r_1} + \mu\left(w_g^{r_1} - w_g^{r_2}\right), \quad (1)$$
$$v_{g+1}^i = w_g^{best} + \mu\left(w_g^{r_1} - w_g^{r_2}\right), \quad (2)$$
$$v_{g+1}^i = w_g^{r_1} + \mu\left(w_g^{r_2} - w_g^{r_3}\right), \quad (3)$$
$$v_{g+1}^i = w_g^{i} + \mu\left(w_g^{best} - w_g^{i}\right) + \mu\left(w_g^{r_1} - w_g^{r_2}\right), \quad (4)$$
$$v_{g+1}^i = w_g^{best} + \mu\left(w_g^{r_1} - w_g^{r_2}\right) + \mu\left(w_g^{r_3} - w_g^{r_4}\right), \quad (5)$$
$$v_{g+1}^i = w_g^{r_1} + \mu\left(w_g^{r_2} - w_g^{r_3}\right) + \mu\left(w_g^{r_4} - w_g^{r_5}\right), \quad (6)$$

where $\mu > 0$ is a real parameter, called mutation constant, which controls the amplification of the difference between two weight vectors and

$$r_1, r_2, r_3, r_4, r_5 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\},$$

are random integers mutually different and different from the running index i. Obviously, the mutation operator results a real weight vector. As our aim is to maintain an integer weight population at each generation, each component of the mutant weight vector is rounded to the nearest integer.

Relation (1) has been introduced as crossover operator for genetic algorithms [4] and is similar to relations (2) and (3). The remaining relations are modifications which can be obtained by the combination of (1), (2) and (3). It is clear that more such relations can be generated using the above ones as building blocks.

To increase further the diversity of the rounded mutant weight vector, the crossover operator is applied. Specifically, for each integer component j ($j = 1, 2, \ldots, N$) of the mutant weight vector v_{g+1}^i, we randomly choose a real number r from the interval $[0, 1]$. Then, we compare this number with ρ (crossover constant), and if $r \leq \rho$ we select as the j-th component of the trial vector u_{g+1}^i the corresponding component j of the mutant vector v_{g+1}^i. Otherwise, we pick the j-th component of the integer target vector w_{g+1}^i. It must be noted that the result of this operation is again an integer vector.

3 Functionality Tests

A set of three learning test problems – XOR, 3-Bit Parity, and the encoder/decoder problems – has been used for testing the functionality, and computer simulations have been developed to study the performance of the DE training algorithms. The simulations have been carried out on a Pentium 133MHz PC IBM compatible using MATLAB version 5.01. For each of the test problems we present a table summarizing the performance of the DE algorithms using different mutation rules. We call DE1 the algorithm that uses relation (1) as mutation operator, DE2 the algorithm that uses relation (2), and so on. The reported parameters for simulations that reached solution are: *min* the minimum number of error function evaluations, *mean* the mean value of error function evaluations, *max* the maximum number of error function evaluations, *s.d.* the standard deviation of error function evaluations, and *succ.* simulations succeeded out of 100 within the generation limit *maxgen*. When an algorithm fails to converge within the *maxgen* limit is considered that it fails to train the FNN and its error function evaluations are not included in the statistical analysis of the algorithms.

For all the simulations we used bipolar input and output vectors and hyperbolic tangent activation functions in both the hidden and output layer neurons. We made no effort to tune the mutation and crossover parameters, μ and ρ respectively. Fixed values ($\mu = 0.5$ and $\rho = 0.7$)

were used instead. The weight population was initialized with random integers from the interval $[-\Delta, \Delta]$. Despite the fact that initial weights with $\Delta > 1$ often helped the algorithms to converge faster, for the simulations we used $\Delta = 1$ because our intention was to train the network with weights as small as possible.

The weight population size NP was chosen to be two times the dimension of the problem, i.e. $NP = 2N$, for all the simulations. Some experimental results have shown that a good choice for NP is $2N \leq NP \leq 4N$. It is obvious that the exploitation of the weight space is more effective for large values of NP, but sometimes more error function evaluations are required. On the other hand, small values of NP make the algorithm inefficient and more generations are required in order to converge to the minimum.

3.1 The Exclusive-OR Problem

The first test problem we will consider is the exclusive-OR (XOR) Boolean function, which is a difficult classification problem and thus a good benchmark. The XOR function maps two binary inputs to a single binary output. A 2-2-1 FNN (six weights, three biases) was used for these simulations and the training was stopped when the value of the error function E, was $E \leq 0.01$ within $maxgen = 100$ generations. The population size was $NP = 18$. The results

Algorithm	min	mean	max	s.d.	succ.
DE1	108	547.9	1332	268.1	91%
DE2	60	180.5	500	91.0	80%
DE3	126	551.7	1656	281.5	95%
DE4	120	271.7	940	133.9	82%
DE5	90	283.6	1530	256.2	81%
DE6	126	720.9	1782	387.2	93%
$NP = 18$, $\mu = 0.5$, $\rho = 0.7$, $maxgen = 100$					

Table 1: Results of simulations for the XOR problem

of the simulation are shown in Table 1. A typical weight vector after the end of the training proccess is $w = (2, -3, -2, 2, 3, 3, -2, -2, 2)$ and the corresponing value of the error function was $E = 0.003$. The six first components of the above vector are the weights and the remaining three are the biases. For this problem DE1, DE3 and DE6 have shown excellent performance. The success rates of all strategies are better than any other well-known continuous weight training algorithm, as far as we know.

3.2 3-Bit Parity

The second test problem is the parity problem, which can be considered as a generalized XOR problem but it is more difficult. The task is to train a neural network to produce the sum, mod 2, of 3 binary inputs – otherwise known as computing the "odd parity" function. We use a 3-3-1 FNN (twelve weights, four biases) in order to train the 3-Bit Parity problem. The initial population consists of 32 weight vectors. The results of the computer simulation are summarized in the Table 2. A typical weight vector after the end of the training proccess is $w = (1, -3, 1, -2, 2, 2, -3, -4, 3, 3, 2, -3, -1, 0, -1, 0)$ and the corresponing value of the error function was $E = 0.009$.

Algorithm	min	mean	max	s.d.	succ.
DE1	660	1721.2	2910	640.4	72%
DE2	150	517.4	1980	298.2	97%
DE3	900	2004.4	2910	561.6	81%
DE4	300	768.2	2040	379.5	99%
DE5	180	732.1	2430	472.9	89%
DE6	600	2115.6	2970	645.4	50%
$NP = 32$, $\mu = 0.5$, $\rho = 0.7$, $maxgen = 100$					

Table 2: Results of simulations for the 3-Bit parity problem

3.3 4-2-4 Encoder/Decoder

The last test problem we considered is the 4-2-4 encoder/decoder. The network is presented with 4 distinct input patterns, each having only one bit turned on. 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 4 patterns in the 2 hidden units and a set of connection weights performing the encoding and decoding operations. This problem was selected because it is quite close to real world pattern classification tasks, where small changes in the input pattern cause small changes in the output pattern [1]. Table 3 summarizes the results.

Algorithm	min	mean	max	s.d.	succ.
DE1	2640	2838.0	3036	280.1	2%
DE2	448	912.7	4096	569.4	84%
DE3	1984	4501.3	6272	1098.6	60%
DE4	512	1026.6	3776	438.6	100%
DE5	246	1192.5	4092	757.2	78%
DE6	3136	4640.0	6272	956.7	18%
$NP = 64$, $\mu = 0.5$, $\rho = 0.7$, $maxgen = 100$					

Table 3: Results of simulations for the 4-2-4 Encoder/Decoder problem

4 Concluding Remarks

In this work, DE based algorithms for integer weight neural network training are introduced. The performance of these algorithms has been examined and simulation results from some classical test problems have been presented. Summarizing the simulations, we can conclude that algorithm DE4 is definitely the best choice for the examined problems. On the other hand, even the algorithm DE1, based on the simple strategy (1), performed well in low dimensional problems.

Customized DE operators have been applied on the population of integer weight vectors, in order to evolve them over time and exploit the weight space as wide as possible. The results indicate that these algorithms are promising and effective, even when compared with other well-known algorithms that require the gradient of the error function and train the network with real weights. These operators have been designed keeping in mind that the resulting integer weights require less bits in order to be stored and the digital arithmetic operations between

them are easier to be implemented in hardware. Furthermore, it is known that the hardware implementation of the backward passes, which compute the gradient of the error function is more difficult. That is why all the proposed algorithms require only forward passes (resulting the value of the error function).

For all the test problems we considered, no choice of the parameters was needed in order to obtain optimal or at least nearly optimal convergence speed. The parameters had fixed values and we made no effort to tune them, but one can try to tune the μ, ρ and NP parameters to achieve better results, i.e. less error function evaluations and/or exhibit higher success rates.

More sophisticated rules must be designed in order to constrain the weights in a specific set of integers and to reduce the amount of memory required for weight storage in digital electronic implementations. Future work also includes evaluation of the generalization performance on well-known classification benchbarks, optimization of the algorithm's performance, and exhausting simulations on bigger and more complex real-life training tasks.

References

[1] S. Fahlman, *An empirical study of learning speed in back-propagation networks*, Technical Report CMU-CS-88-162, Carnegie Mellon University, Pittsburgh, PA 15213, September 1988.

[2] A.H. Khan and E.L. Hines, *Integer-weight neural nets*, Electronics Letters, **30**, 1237-1238, (1994).

[3] A.H. Khan, *Feedforward Neural Networks with Constrained Weights*, Ph. D. dissertation, Univ. of Warwick, Dept. of Engineering, (1996).

[4] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, (1996).

[5] R. Storn and K. Price, *Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous spaces*, Journal of Global Optimization, **11**, 341–359, (1997).