Training Neural Networks with 3–bit Integer Weights

V.P. Plagianakos
University of Patras
Department of Mathematics,
U.P. Artificial Intelligence
Research Center (UPAIRC),
GR-26500 Patras, Greece.
e-mail: vpp@math.upatras.gr

M.N. Vrahatis
University of Patras
Department of Mathematics,
U.P. Artificial Intelligence
Research Center (UPAIRC),
GR-26500 Patras, Greece.
e-mail: vrahatis@math.upatras.gr

Abstract

In this work we present neural network training algorithms, which are based on the differential evolution (DE) strategies introduced by Storn and Price [Journal of Global Optimization. **11**:341–359, 1997]. These strategies are applied to train neural networks with 3–bit integer weights. Integer weight neural networks are better suited for hardware implementation than their real weight analogous. Moreover, we constrain the weights and biases in the range $[-3, 3]$, thus, they can be represented by just 3 bits. This property reduces the amount of memory required and simplifies the digital multiplication operation.

Our intention is to present a broad picture of the behavior of this class of evolution algorithms in this difficult task. Simulation results from classical benchmarks show that these methods are promising, fast, and reliable.

1 INTRODUCTION

Artificial Feedforward Neural Networks (FNNs) have been widely used in many application areas in recent years and have shown their strength in solving hard problems in Artificial Intelligence. Although many different models of neural networks have been proposed, multilayered FNNs are the most common. FNNs consist 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 its 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 high 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, are 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 be achieved.

Another advantage of the FNNs with integer weights is their immunity to noise in the training data. Such networks only capture the main feature of the training data. Low amplitude noise that possibly contaminates the training data cannot perturb the discrete weights, because those weights require relative large variations to jump from one integer value to another.

Moreover, neural networks with integer weights in the range $[-3, 3]$ can be represented by just 3 bits. This property reduces the amount of memory required for weight storage in digital electronic implementations. Additionally, it simplifies the digital multiplication operation, as multiplying any number with a 3–bit integer requires a maximum of three basic instructions, namely: one shift, one addition, and one sign change. Finally, if inputs are restricted to the set $\{-1, 1\}$ (bipolar inputs), the neurons in the first hidden layer require only sign change for multiplication operations, and only integer addition.

The efficient supervised training of FNNs, i.e. the incremental adaptation of the connection weights that propagate information between the neurons, is a subject of considerable ongoing research and numerous algorithms have been proposed to this end. The majority of those algorithms use the negative of the gradient of the error function, $-\nabla E(w)$, as their descent

direction. The gradient $\nabla E(w)$ can be computed by the BackPropagation of the error through the layers of the network. This calculation, however, is computationally expensive and difficult to be implemented in hardware. In this contribution, we propose a new class of training algorithms that do not need the gradient of E.

Formally, a typical FNN consists 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. The 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 i-th neuron at the $(l-1)$ layer to the j-th neuron at the l-th layer, y_i^l is the output of the ith neuron belonging to the l-th 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 *3–bit integer* vector of the 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 the pattern t. Minimization of E is attempted by using a training algorithm to update the weights. Efficient training algorithms have been proposed for trial and error based training, but it is difficult to use them when training with discrete weights [1, 2].

In this work a differential evolution approach, as explained in Section 2, has been utilized to train a neural network with 3–bit 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. In Section 3, in addition to the speed and robustness, we also evaluate the generalization capabilities of 3–bit integer weight neural networks, by testing them on the MONK's problems [6]. The final section contains concluding remarks and a short discussion for future work.

2 TRAINING NEURAL NETWORKS WITH INTEGER WEIGHTS

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 that DE has good convergence properties and outperforms other evolutionary algorithms.

In order to apply DE to neural network training with 3–bit 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 $[-3, 3]$ 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. Moreover, we force the new vectors to be in the range $[-3, 3]^N$. This operation is called *mutation*. The outcoming 3–bit 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, in the range $[-3, 3]^N$. 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.

2.1 THE MUTATION OPERATOR

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:

$$v_{g+1}^i = w_g^{r_1} + \mu \left(w_g^{r_1} - w_g^{r_2} \right), \quad (1)$$

$$v_{g+1}^i = w_g^{\text{best}} + \mu \left(w_g^{r_1} - w_g^{r_2} \right), \quad (2)$$

$$v_{g+1}^i = w_g^{r_1} + \mu \left(w_g^{r_2} - w_g^{r_3} \right), \quad (3)$$

$$v_{g+1}^i = w_g^i + \mu \left(w_g^{\text{best}} - w_g^i \right) + \mu \left(w_g^{r_1} - w_g^{r_2} \right), \quad (4)$$

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

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

where w_g^{best} is the best member of the previous generation, $\mu > 0$ 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. Additionally, if the mutant vector is not in the range $[-3, 3]^N$, we take:

$$v_{g+1}^i = \text{sign}(v_{g+1}^i) \times \left(\left| v_{g+1}^i \right| \mod 4 \right).$$

Relation (1) has been introduced as crossover operator for genetic algorithms [3] 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. In a previous work [4], we have shown that the above relations can efficiently be used to train FNNs with arbitrary integer weights.

2.2 THE CROSSOVER OPERATOR

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 a 3–bit integer vector.

3 FUNCTIONALITY TESTS

Two classical learning test problems – the eXclusive OR (XOR) and the 3-Bit Parity problems – have been used for testing the functionality, and computer simulations have been developed to study the performance of the DE training algorithms. In order to examine the generalization behavior of these algorithms, we have tested the best of them on the well known MONK's problems [6].

The simulations have been carried out on a IBM PC 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 have 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 1000 within the generation limit *maxgen*. When an algorithm fails to converge within the *maxgen* limit, it is considered that it fails to train the FNN and its error function evaluations are not included in the statistical analysis of the algorithms. We must note here that a key feature of the DE algorithms is that only error function values are needed. No gradient information is required, so there is no need of backward passes.

For all the simulations we have 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 $\rho = 0.7$) have been used instead. The weight population has been initialized with random integers from the interval $[-3, 3]$.

The weight population size *NP* has been chosen to be twice 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.

For the test problems considered, no choice of the parameters has been needed in order to obtain optimal or at least nearly optimal convergence speed. The parameters we have used have fixed values and we have made no effort to tune them. Of course, 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.

3.1 THE EXCLUSIVE-OR PROBLEM

The first test problem we will consider is the exclusive-OR (XOR) Boolean function problem, which historically has been considered as a good test of a network model and learning algorithm. The XOR function maps two binary inputs to a single binary output. This simple Boolean function is not linearly separable and thus requires the use of extra hidden units to learn the task. Moreover, it is sensitive to initial weights as well as to learning rate variations and presents a multitude of local minima with certain weight vectors. A 2-2-1 FNN (six weights, three biases) has been used for these simulations and the training has been stopped when the value of the error function E, has been $E \leq 0.1$ within $maxgen = 100$ generations. The population size is *NP*=18. The results of the simulation are shown

Table 1: Results of simulations for the XOR problem

Algorithm	min	mean	max	s.d.	succ.
DE1	54	191.9	810	89.7	63.0%
DE2	90	836.3	1782	371.7	93.5%
DE3	90	300.5	1584	171.2	82.5%
DE4	54	364.5	1676	222.6	93.4%
DE5	36	1047.8	1782	422.6	63.0%
DE6	54	931.5	1782	431.1	73.1%
NP=18, $\mu = 0.5$, $\rho = 0.7$, *maxgen*=100					

in Table 1. A typical weight vector after the end of the training process is $w = (3, 3, 2, 3, 2, -2, 1, -3, -2)$ and the corresponding value of the error function is $E = 0.0221$. The six first components of the above vector are the weights and the remaining three are the biases.

For this problem DE2, DE3 and DE4 have shown excellent performance. The success rates of all strategies are better than other well-known continuous weight training algorithm, such as BackPropagation (BP), adaptive BP or BP with momentum.

3.2 3-BIT PARITY

The second test problem is the parity problem, which can be considered as a generalized XOR problem but 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 process is $w = (3, 3, 2, 3, -1, -1, 2, -2, -2, -3, 3, -3, 1, 0, 1, 1)$ and the corresponding value of the error function is $E = 0.0257$.

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

Algorithm	min	mean	max	s.d.	succ.
DE1	96	809.2	2016	313.9	88.1%
DE2	704	1966.9	3072	769.2	1.5%
DE3	320	1123.4	3168	461.0	97.7%
DE4	160	2072.1	3168	631.9	75.5%
DE5	1344	2057.1	3072	703.9	0.7%
DE6	160	1890.0	3168	907.7	3.2%
NP=32, $\mu = 0.5$, $\rho = 0.7$, *maxgen*=100					

In a previous work [4], we have shown that DE3 and DE4 are the more suitable algorithms for integer weight training. This is also evident form the results shown in Table 2. In this problem only the DE1, DE3 and DE4 algorithms exhibit good performance.

3.3 GENERALIZATION PERFORMANCE

In order to evaluate the generalization performance of the DE algorithms we have tested the best of them (namely the DE3 and DE4) on the MONK's problems. These are binary classification tasks which have been used for comparing the generalization performance of learning algorithms. These problems rely on the artificial robot domain, in which robots are described by six different attributes. Each problem is given by a logical description of the class, as shown below:

MONK-1: (*Attribute1* = *Attribute2*) OR (*Attribute5* = 1). This problem is in standard Disjunctive Normal Form (DNF). 124 examples have been selected randomly from the data set for training, while the remaining 308 have been used for the generalization testing. There are no misclassifications.

MONK-2: (Only two attributes = 1). This problem is similar to the parity problem mentioned above and is difficult to describe in DNF or Conjunctive Normal Form (CNF). 169 examples have been randomly selected from the data set for training, while the rest have been used for testing. Again, there is no noise.

MONK-3: (*Attribute5* = 3 AND *Attribute4* = 1) OR (*Attribute5* $\neq$ 4 AND *Attribute2* $\neq$ 3) with added noise. This problem is also in DNF but with 5% deliberate misclassifications in the training set, which consists of 122 examples. The remaining 310 examples have been used for testing.

Each one of the six attributes can have one of 3, 3, 2, 3, 4, and 2 values, respectively, which results 432 possible combinations that constitute the total data set (see [6], for details). Finally, each possible value for every attribute is assigned a single bipolar input, resulting 17 inputs.

We have tested DE3 and DE4 against the BackPropagation (BP), the BackPropagation with Weight Decay (BPWD), and the Cascade Correlation (CC) algorithms. In Table 3 we exhibit the comparative results on the MONK's problems. It is clear from Table 3 that the DE algorithms generate FNNs, which are at least as capable as the best generated by real-weight learning algorithms. Those networks, in all the MONK's problems, seem to have learned the concept embedded in the training data. This is more evident in MONK-3, where there are 5% deliberate misclassifications and the networks generated by BP, BPWD, and CC seem to fail to capture the concept embedded in the training data, and fit to the noise instead.

Table 3: Comparison of generalization performance on the MONK's problems

Algorithm	MONK-1	MONK-2	MONK-3
BP	100%	100%	93.1%
BPWD	100%	100%	97.2%
CC	100%	100%	97.2%
DE3	100%	100%	100%
DE4	100%	100%	100%

The topology of the trained networks is shown in Table 4. It is known that the best generalizers are neither too complex nor too simple; they exactly match the complexity of the embedded in the training data concept. We think that the reason why our algorithms, in general, need a bigger network in order to generate FNNs with good generalization capabilities is that more integers than real numbers are needed in order to match the complexity of the given problem.

Table 4: Network configuration for the MONK's problems

Algorithm	MONK-1	MONK-2	MONK-3
BP	17:3:1	17:2:1	17:4:1
BPWD	17:2:1	17:2:1	17:2:1
CC	17:1:1	17:1:1	17:3:1
DE3	17:4:1	17:4:1	17:3:1
DE4	17:4:1	17:4:1	17:3:1

At the end of this paper, we present three tables with the integer weights of the networks trained using DE3. Similar networks are generated by DE4.

4 CONCLUDING REMARKS

In this work, DE based algorithms for 3–bit integer weight neural network training are introduced. Customized DE operators have been applied on the population of 3–bit integer weight vectors, in order to evolve them over time and exploit the weight space as wide as possible. 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 the DE3 and DE4 algorithms are definitely the best choices. On the other hand, even the algorithm DE1, based on the simple strategy (1), performed very well.

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 in the value of the error function). Moreover, we have tested the generalization capabilities of the networks generated by DE3 and DE4. Both algorithms have exhibited excellent performance and have outperformed other well known real weight learning algorithms.

References

[1] A.H. Khan (1996). *Feedforward Neural Networks with Constrained Weights*. Ph.D. Thesis, Univ. of

Warwick, Dept. of Engineering.

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

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

[4] V.P. Plagianakos, D.G. Sotiropoulos and M.N. Vrahatis (1998). Integer Weight Training by Differential Evolution Algorithms, In N.E. Mastorakis (ed.), 327–331. *Recent Advances in circuits and systems*, World Scientific.

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

[6] S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S.E. Fahlman, D. Fisher, R. Hamann, K. Kaufmann, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek and J. Zhang (1991). *The MONK's Problems: A performance comparison of different learning algorithms*, Technical Report, Carnegie Mellon University, CMU-CS-91-197.

Table 5: MONK's problem #1: weights and biases

From node	**To node**				
	Hid1	Hid2	Hid3	Hid4	Out
In1	3	−3	3	−1	
In2	2	−1	2	3	
In3	3	−2	−3	−2	
In4	2	−3	3	2	
In5	1	1	2	3	
In6	1	−1	−2	1	
In7	−3	3	−3	3	
In8	−2	3	−3	3	
In9	2	2	−3	3	
In10	0	2	−2	3	
In11	1	2	−3	3	
In12	1	−3	−3	3	
In13	2	2	−3	−3	
In14	0	2	−3	−3	
In15	2	2	−3	−3	
In16	−3	−3	−3	2	
In17	−1	−3	−2	2	
Bias	−2	3	3	−3	
Hid1					−3
Hid2					−1
Hid3					−3
Hid4					2
Bias					3

Table 6: MONK's problem #2: weights and biases

From node	**To node**				
	Hid1	Hid2	Hid3	Hid4	Out
In1	−2	−1	−3	−2	
In2	3	2	−2	3	
In3	2	2	−2	3	
In4	1	1	−3	−2	
In5	2	2	−2	3	
In6	3	2	−2	3	
In7	2	2	2	−3	
In8	1	−3	3	2	
In9	3	1	2	−2	
In10	2	2	3	3	
In11	1	1	3	3	
In12	−2	−3	2	−2	
In13	−1	3	3	3	
In14	−2	0	3	3	
In15	−1	2	3	3	
In16	2	−1	2	−3	
In17	2	2	3	2	
Bias	−1	0	3	−3	
Hid1					0
Hid2					0
Hid3					2
Hid4					−2
Bias					0

Table 7: MONK's problem #3: weights and biases

From node	**To node**			
	Hid1	Hid2	Hid3	Out
In1	0	0	1	
In2	−1	0	1	
In3	−1	0	−1	
In4	0	−2	0	
In5	−2	−2	2	
In6	−1	2	0	
In7	1	1	−2	
In8	−2	1	1	
In9	−2	0	0	
In10	−1	2	2	
In11	−2	2	1	
In12	1	−1	1	
In13	−2	−1	−3	
In14	1	−3	0	
In15	0	3	1	
In16	2	0	0	
In17	0	0	1	
Bias	0	1	−2	
Hid1				0
Hid2				−2
Hid3				0
Bias				2