
Natural Computing **1:** 307–322, 2002.

Parallel evolutionary training algorithms for "hardware-friendly" neural networks

VASSILIS P. PLAGIANAKOS and MICHAEL N. VRAHATIS
Department of Mathematics and Artificial Intelligence Research Center – UPAIRC, University of Patras, GR-26110 Patras, Greece (E-mail: {vpp,vrahatis}@math.upatras.gr)

Abstract. In this paper, Parallel Evolutionary Algorithms for integer weight neural network training are presented. To this end, each processor is assigned a subpopulation of potential solutions. The subpopulations are independently evolved in parallel and occasional migration is employed to allow cooperation between them. The proposed algorithms are applied to train neural networks using threshold activation functions and weight values confined to a narrow band of integers. We constrain the weights and biases in the range [–3, 3], thus they can be represented by just 3 bits. Such neural networks are better suited for hardware implementation than the real weight ones. These algorithms have been designed keeping in mind that the resulting integer weights require less bits to be stored and the digital arithmetic operations between them are easier to be implemented in hardware. Another advantage of the proposed evolutionary strategies is that they are capable of continuing the training process "on-chip", if needed. Our intention is to present results of parallel evolutionary algorithms on this difficult task. Based on the application of the proposed class of methods on classical neural network problems, our experience is that these methods are effective and reliable.

Key words: "hardware-friendly" implementations, integer weight neural networks, "on-chip" training, parallel differential evolution algorithms, threshold activation functions

Abbreviations: FNN – feedforward neural network; PEA – parallel evolutionary algorithm; EHW – evolvable hardware; ES – evolution strategy; DE – differential evolution; PDE – parallel differential evolution

ACM Computing Classification (1998): I.2.6, C.1.3, F.1.1, G.1.6, F.1.2, G.1.0

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 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. Hardware-friendly algorithms are essential to ensure the functionality and cost effectiveness of the hardware implementation. Moreover, the need for hardware-friendly algorithms, which have the ability to cope with time-varying problems and real-time timing constraints, has been recently increased. The evolvable hardware (EHW) (Higuchi et al., 1992; Higuchi et al., 1994) designed for such practical industrial applications. In the conventional hardware design, it is necessary to have in advance all the specifications of the hardware functions. To alleviate this problem (especially in time-varying or unknown environments), EHW design has been employed (see for example (Yao and Higuchi, 1999)). The method of EHW design is able to change dynamically the hardware configurations according to an Evolution Strategy (ES) (Beyer and Schwefel, 2002; Schwefel, 1995). To improve EHW's performance, these reconfigurations do not need in advance known specifications and can continue on-line, if necessary.

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. Additionally, the use of threshold activation functions for all the hidden and output neurons, greatly reduces the complexity of the hardware implementation, because there is no need to design and implement complicated non-linear activation functions. Another advantage of the FNNs with integer weights and threshold activation functions is that the trained neural network is to some extend immune 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 set cannot perturb the discrete weights, because those networks require relatively large variations to "jump" from one integer weight value to another.

Mathematical operations that are easy to implement in software might often be very burdensome in the hardware and therefore more costly. To this end, we focus on neural networks having integer weights, constrained in the ranges [–3, 3], which correspond to 3-bit integer representation of the weights. This property reduces the amount of memory required for weight storage in digital electronic implementations. Additionally, it simplifies the digital multiplication operation, since multiplying any number with a 3-bit integer requires only one sign change, one one-step left shift and one addition.

Finally, if inputs are restricted to the set {–1, 1} (bipolar inputs), the neurons in the first hidden layer require only sign changes during multiplication operations, and only integer additions.

It is evident that the problem of neural network training using integer weights is related to the integer programming problem. Early approaches in the direction of Evolutionary Algorithms (EAs) for integer programming can be found in (Gall, 1966; Kelahan and Gaddy, 1978). They proposed random search methods on integer spaces in the spirit of a (1 + 1)-ES (Beyer adn Schwefel, 2002; Schwefel, 1995). More recently, an interesting construction of a mutation distribution for unbounded integer search spaces is proposed in (Rudolph, 1994) resulting to an efficient (μ, λ)-ES for integer programming. In our approach we utilize a Parallel Evolutionary Algorithm (PEA). The proposed algorithm is based on the recently proposed differential evolution method (Storn, 1999; Storn and Price, 1997). Our experimental results show that this algorithm is an efficient and effective algorithm for neural network training with integer weights.

The paper is organized as follows: in the next section we briefly review the basics of neural network training and formulate the problem. The proposed PEA approach, as explained in Section 3, has been utilized to train neural networks with 3-bit integer weights and threshold activation functions, suitable for hardware implementation. A brief overview of the chosen ES is also presented. In Section 4 techniques for training neural networks with threshold activation functions are proposed. Experiments and computer simulation results are presented in Section 5. The final section contains concluding remarks and a short discussion.

2. Neural network training

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 (Rumelhart et al., 1994; Magoulas et al., 1997) of the error through the layers of the network. This calculation, however, is computationally expensive and difficult to be implemented in hardware. In this paper, a new class of training algorithms that do *not* need the gradient of E and train networks with threshold units is proposed. Other algorithms that train neural networks with threshold units need the learning task to be static, i.e., not to change over time, in order to train the network "off-line" in a software simulation and later transfer it to

the hardware (Corwin et al., 1994; Magoulas et al., 1997). But many real-life applications may not be static, i.e., input data may continue to change even after the hardware implementation. In such cases an algorithm capable to continue training "on-chip" is needed. The advantage of the proposed strategies is that they are capable of continuing the training process in hardware, when threshold activation functions have been used.

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

$$net_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1} + \theta_j^l, \qquad y_j^l = f^l\left(net_j^l\right),$$

where $w_{ij}^{l-1,l}$ is the integer 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 i-th neuron belonging to the l-th layer, θ_j^l denotes the integer bias of the j-th neuron at the l-th layer, and f is the 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 the corresponding 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 (Khan, 1996; Khan and Hines, 1994).

3. The proposed parallel evolutionary algorithm

In a recent work, Storn and Price (Storn and Price, 1997) have presented a novel minimization method, called Differential Evolution (DE), which has been designed to handle nondifferentiable, 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 well known evolutionary algorithms (Storn, 1999; Storn and Price, 1997). EAs, as well as DEs, are easily parallelized due to the fact that each member of the population is evaluated individually (Schwefel, 1995). The only phase of the algorithm which requires communication with other individuals is in reproduction, and this too occurs in parallel for pairs of individuals (Michalewicz and Fogel, 2000; Schwefel, 1995). Thus, there are two typical models for EA parallelization. The first uses fine grained parallelism, so each individual is represented by a processor. This creates certain problems when the number of processors available is limited or when the individual's fitness to reproduce needs to be evaluated over the whole population. The second model, which is actually used in this paper, maps an entire subpopulation to a processor. Thus each subpopulation evolves independently towards a solution. This allows each subpopulation to develop its own solution uniquely. Then, the best individual of each subpopulation is propagated to other subpopulations, according to the selected topology. This operation is called "migration" (Rudolph, 1991). This model is called the Parallel Evolutionary Algorithm (PEA). The topology of the proposed PEA is a ring, i.e., the best individuals from each subpopulation are allowed to migrate to the next subpopulation of the ring. This concept reduces the migration between the subpopulations and consequently the messages between the processors. The migration of the best individuals is controlled by the migration constant, $\phi \in (0, 1)$. At each iteration, a random number from the interval (0, 1) is uniformly chosen and compared with the migration constant. If the migration constant is bigger, then the best individuals of each subpopulation migrate and take the place of a randomly selected individual (different from the best one) in the next subpopulation; otherwise no migration is permitted. We have experimentally found that a migration constant, $\phi = 0.1$, is a good choice, since it allows

each subpopulation to evolve for some iterations before the migration phase actually occur.

To apply Parallel DE (PDE) to neural network training with 3-bit integer weights, we start with a specific number of subpopulations, each one initialized with NP, N-dimensional integer weight vectors, and evolve them over time. NP is fixed throughout the training process. All the weight subpopulations are initialized with random integers from the interval [–3, 3], following a uniform probability distribution. At each iteration, called *generation*, all the subpopulations are evolved independently in parallel, until a migration phase is decided. After the migration of the best individuals, the new subpopulations continue to evolve as before.

Let us now focus on a subpopulation. In each subpopulation, new weight vectors are generated by combining weight vectors that are 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 in our context can be referred as *mutation*. The outcoming 3-bit integer weight vectors are then mixed with another predetermined integer weight vector – the *target* weight vector – and this operation can be called as *recombination* (see Section 3.1). 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 operation can be referred as *selection*.

A high-level description of the usage of the above mentioned operators is given below (for one generation):

Step 1: **Do** for each weight_vector
Step 2: mutant_vector := MUTATION(weight_vector)
Step 3: trial_vector := RECOMBINATION(mutant_vector)
Step 4: **If** E(trial_vector) $\leqslant$ E(weight_vector)
Step 5: weight_vector := trial_vector
Step 6: **EndIf**
Step 7: **EndDo**

To prevent a vector from surviving indefinitely, we employ the concept of aging (Storn, 1999). To this end, each vector is randomly assigned a maximum age, i.e., an integer from the interval $[\alpha, \beta]$, where α and β are the minimum and the maximum possible age, respectively. At each iteration, the age of each vector is increased by one, and if it exceeds its maximum age then the individual "dies". This individual is then replaced by another vector randomly chosen from the current subpopulation. Note that it is desirable not to eliminate the best individual of the subpopulation. We now briefly review the two basic PDE variation operators used for integer weight FNN training.

3.1 *Variation operators*

The first PDE 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:

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

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

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

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

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

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 in such away to avoid the search to be stagnated and

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

are random integers mutually different and different from the running index i. Obviously, the mutation operator results in 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:

$$v_{g+1}^i = \text{sgn}(v_{g+1}^i) \cdot \left(\left| v_{g+1}^i \right| \bmod 4 \right),$$

where sgn is the well known triple valued sign function. Relation (1) is similar to the intermediary recombination operator of ES, and relations (2) and (3) derive from it. 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 (Plagianakos and Vrahatis, 1999; Plagianakos and Vrahatis, 2000), it has been shown that the above relations can efficiently be used to train FNNs with arbitrary as well as constrained integer weights.

To increase further the diversity of the rounded mutant weight vector, the recombination operator is applied. Specifically, for each integer component j (j =1, 2, ..., 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 (recombination constant), and if $r \leqslant \rho$ 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^i_{g+1}. It must be noted that the result of this operation is again a 3-bit integer vector.

4. Using threshold activation functions

The proposed class of algorithms does not need the activation function to be differentiable and is suitable for training with threshold units (Plagianakos and Vrahatis, 2000). In the first phase of our approach, the PDE algorithms are used to train a neural network "off-line", using sigmoid activation functions, such as:

$$\begin{aligned} f_1(x) &= \frac{2}{1+e^{-\lambda x}} - 1, \\ f_2(x) &= \frac{1}{1+e^{-\lambda x}}, \\ f_3(x) &= \tanh \frac{\lambda x}{2}, \end{aligned}$$

where λ is the gain parameter. This seems to be a good practice since the network is trained much faster with sigmoid functions. In the second phase we alter the gain of the sigmoid function in such a way that allows a mapping to a threshold unit network.

Specifically, when the inputs are correctly classified and the network error is relatively small, the value of λ is increased in the sequence (1, 10, 20, 30, 40, 50, ∞). Additional training might be necessary after each increase of λ. That justifies the additional iterations needed to train an FNN, using only threshold activation functions. This procedure is analogous to taking the limit of the sigmoid function as the gain parameter λ goes to infinity. Finally, the trained network uses only threshold activation functions and thus the complexity of the hardware implementation is greatly reduced. If new input data are introduced, training *can* be continued sequentially or in parallel "on-chip", using the proposed algorithms.

5. Experimental results

Three classical learning test problems – the eXclusive–OR (XOR), the 3-Bit Parity and the Encoder/Decoder – have been used for testing the functionality, and parallel computer simulations have been developed to study the performance of the PDE training algorithms. We call PDE1 the algorithm that uses relation (1) as mutation operator, PDE2 the algorithm that uses relation (2), and so on. For all the simulations bipolar input and output vectors have

been used. Tables 1, 3, 5 summarize the performance of the PDE algorithms using different mutation rules when sigmoid activation functions are used. Hyperbolic tangent activation functions in both the hidden and output layer neurons have been used. In Tables 2, 4, 6 we exhibit the performance of the PDE algorithms to the same test problems, when the training has been performed as described in Section 4 and leads to a trained network that uses only threshold activation functions.

For each problem we have conducted 100 simulations and the reported parameters in the following tables for simulations that have reached solution are: *min* the minimum number, *mean* the mean value, *max* the maximum number, and *s.d.* the standard deviation of error function evaluations. When an algorithm fails to converge, 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 PDE algorithms is that *only* error function values are needed. No gradient information is required, so there is no need of backward passes. For the test problems considered, we made no effort to tune the mutation, recombination and migration constants, μ, ρ and ϕ respectively, to obtain optimal or at least nearly optimal convergence speed. Default fixed values ($\mu = 0.5$, $\rho = 0.7$ and $\phi = 0.1$) have been used instead. Smaller values of ϕ can further reduce the messages between the processors, but may result in rare and inefficient migrations. It is obvious that one can try to fine-tune the μ, ρ, ϕ and NP parameters to achieve better results, i.e., less error function evaluations and/or exhibit higher success rates. The weight subpopulations have been initialized with random integers from the interval [–3, 3] and the total population size $3NP$ has been divided equally to 3 subpopulations, each having NP individuals. Regarding the total population size, experimental results have shown that a good choice is $2N \leqslant 3NP \leqslant 4N$. 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 render the algorithm inefficient and more generations are required to converge to the minimum.

5.1 *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. A 2–2–1 FNN (six weights and three biases, dimension of the problem $N = 9$) has been used for these simulations and the training has been stopped when the value of the error function E, has been $E \leqslant 0.1$. The size of each subpopulation was $NP = 10$. The low and high bound of the age of each individual, were $\alpha = 20$ and $\beta = 30$ respectively. A

typical 3-bit 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. Tables 1 and 2 exhibit the simulation results when sigmoid and threshold activation functions were used.

Table 1. Results of simulations for the XOR problem using sigmoid activation functions.

Algorithm	min	mean	max	s.d.	Success
PDE1	40	238.4	750	136.3	100%
PDE2	130	720.1	1840	352.6	100%
PDE3	80	342.2	1090	186.1	100%
PDE4	50	395.6	1080	218.5	100%
PDE5	140	1209.7	3360	661.5	100%

Table 2. Results of simulations for the XOR problem using threshold activation functions.

Algorithm	min	mean	max	s.d.	Success
PDE1	360	590.4	1150	159.5	100%
PDE2	420	1197.6	5340	614.0	100%
PDE3	390	651.3	1250	177.5	100%
PDE4	380	746.9	1480	225.6	100%
PDE5	490	1473.8	3790	586.2	100%

5.2 *The 3-bit parity problem*

The second test problem is the 3-bit 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 – also known as computing the "odd parity" function. We use a 3–3–1 FNN (twelve weights and four biases, dimension of the problem N = 16) in order to train the 3-Bit Parity problem. Each subpopulation consists of 11 weight vectors. The maximum age of each individual has been randomly selected from the interval $[\alpha, \beta]$, where α = 50 and β = 100. A typical 3-bit 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. Tables 3 and 4 illustrate the results for this problem.

Table 3. Results of simulations for the 3-bit parity problem using sigmoid activation functions.

Algorithm	min	mean	max	s.d.	Success
PDE1	275	1272.9	3949	619.1	82%
PDE2	1353	3562.7	8525	1367.8	86%
PDE3	198	1473.0	6457	873.3	91%
PDE4	264	2227.3	5104	903.5	99%
PDE5	1430	4829.6	9306	1598.2	91%

Table 4. Results of simulations for the 3-bit parity problem using threshold activation functions.

Algorithm	min	mean	max	s.d.	Success
PDE1	561	3022.3	12078	3523.2	100%
PDE2	1562	5222.3	25377	3757.7	100%
PDE3	660	2238.8	11847	2100.7	100%
PDE4	1419	3147.7	11517	1742.4	100%
PDE5	2387	6868.2	35310	5473.2	100%

5.3 *4–2–4 encoder/decoder*

The last test problem we considered is the 4–2–4 encoder/decoder (sixteen weights and six biases, dimension of the problem $N = 22$). 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 particular encoding is considered to be "tight", since the number of the hidden nodes equals the base 2 logarithm of the input nodes ($\log_2 4 = 2$). This problem has been 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. The size of each subpopulation was $NP = 20$. The low and high bound of the age of each individual, were $\alpha = 50$ and $\beta = 200$ respectively. A typical 3-bit weight vector is $w = (0, 2, -2, 3, -3, -3, 2, 3, -3, -3, 2, -3, -2, 2, 3, 2, 1, 0, -3, -3, -2, -2)$ and the corresponding value of the error function is $E = 0.0459$. Simulation results are exhibited in Tables 5 and 6.

Table 5. Results for the encoder/decoder problem using sigmoid activation functions.

Algorithm	min	mean	max	s.d.	Success
PDE1	330	1614.8	4686	868.5	100%
PDE2	3960	8160.6	13376	2160.5	100%
PDE3	308	1428.2	4004	660.9	100%
PDE4	660	4540.5	8514	1505.4	100%
PDE5	7260	13110.9	20636	3092.4	100%

Table 6. Results for the encoder/decoder problem using threshold activation functions.

Algorithm	min	mean	max	s.d.	Success
PDE1	990	2520.8	23260	2326.4	100%
PDE2	4796	8724.9	16588	2264.6	100%
PDE3	1034	2104.5	4664	680.0	100%
PDE4	1870	4778.1	9724	1278.0	100%
PDE5	6072	14070.3	20746	2795.4	100%

5.4 *Monk's problem and generalization results*

In addition to training speed and efficiency, we have also evaluated the generalization performance of the PDE algorithms. To this end, we have tested the best of them (PDE3 and PDE4) on the MONK's problems. These are difficult 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 (Thrun et al., 1991), for details). Finally, each possible value for every attribute is assigned a single bipolar input, resulting 17 inputs.

We have tested PDE3 and PDE4 against the BackPropagation (BP), the BackPropagation with Weight Decay (BPWD), and the Cascade Correlation (CC) algorithms. In Table 7 we exhibit the comparative results on the MONK's problems.

It is clear from Table 7 that the PDE 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 7. 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%
PDE3	100%	100%	100%
PDE4	100%	100%	100%

The topology of the trained networks is shown in Table 8. The dimension for each of the MONK's problems is $N = 77$, $N = 77$, and $N = 58$, respectively. 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 to match the complexity of the given problem (Khan, 1996).

Table 8. 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
PDE3	17:4:1	17:4:1	17:3:1
PDE4	17:4:1	17:4:1	17:3:1

6. Concluding remarks and discussion

In this paper, Parallel Differential Evolution algorithms for 3-bit integer weight neural networks with threshold activation functions are studied. This is an interesting kind of neural networks, because the amount of memory required for the storage of their weights is significantly reduced compared to networks with real weights and non-linear (sigmoid) activation functions and the digital arithmetic operations required are simplified. Moreover, this kind of networks are based on neurons whose output can be in a particular state and are important, since they can handle many of the inherently binary tasks that neural networks are used for. Their internal representations are clearly interpretable, they are computationally simpler to understand than networks with sigmoid units and provide a starting point for the study of the neural network properties (Boutsinas and Vrahatis, 2001; Magoulas et al., 1997). Furthermore, the training procedure can continue on-chip, if the environment has changed.

Customized differential evolution operators have been applied on subpopulations of 3-bit integer weight vectors, in order to evolve them over time in parallel and explore the constrained weight space as wide as possible. The proposed algorithms require only forward passes resulting in the value of the error function, since the hardware implementation of the backward passes, which compute its gradient is more difficult. The performance of these algorithms has been examined and simulation results from some classical test problems have been presented. The results suggest that the PDE algorithms are promising, effective and suitable for integer weight training, when sigmoid or threshold activation functions are used. The success rates of some of these strategies are better than other well-known continuous weight training algorithm that require the gradient of the error function, such as BackPropagation (BP), adaptive BP or BP with momentum. Summarizing, we have concluded that the PDE3 and PDE4 algorithms seem to be the best choices for the problems tested. On the other hand, algorithm PDE1, based on a simple strategy, has performed remarkably well.

Acknowledgments

The authors wish to thank Prof. Hans-Paul Schwefel for his very constructive comments in the early draft of this paper. This material was partially supported by the Deutsche Forschungsgemeinschaft (DFG) as a part of the collaborative research center "Computational Intelligence" (SFB 531).

Part of this work was done while the author was at the Department of Computer Science XI, University of Dortmund, D-44221 Dortmund, Germany.

References

Beyer H-G and Schwefel H-P (2002) Evolution Strategies: A comprehensive introduction. Natural Computing (to appear)

Boutsinas B and Vrahatis MN (2001) Artificial nonmonotonic neural networks. Artificial Intelligence 132: 1–38

Corwin EM, Logar AM and Oldham WJB (1994) An Iterative Method for Training Multilayer Networks with Threshold Functions. IEEE Transactions on Neural Networks 5: 507–508

Gall DA (1996) A practical multifactor optimization criterion. In: Lavi A and Vogl TP (eds) Recent Advances in Optimization Techniques, pp. 369–386. Wiley, New York

Higuchi T, Niwa T, Tanaka T, Iba H, Garis H and Furuya T (1992) Evolvable hardware with genetic learning. Simulation of Adaptive Behavior, MIT Press

Higuchi T, Iba H and Manderick B (1994) Evolvable hardware with genetic learning. In: Kitano H (ed) Massively Parallel Artificial Intelligence. MIT Press

Kelahan RC and Gaddy JL (1978) Application of the adaptive random search to discrete and mixed integer optimization. International Journal for Numerical Methods in Engineering 12: 289–298

Khan AH (1996) Feedforward Neural Networks with Constrained Weights. Ph.D. Thesis, Univ. of Warwick, Dept. of Engineering

Khan AH and Hines EL (1994) Integer-weight neural nets. Electronics Letters 30: 1237–1238

Magoulas GD, Vrahatis MN and Androulakis GS (1997) Effective back-propagation with variable stepsize. Neural Networks 10: 69–82

Magoulas GD, Vrahatis MN, Grapsa TN and Androulakis GS (1997) A training method for discrete multilayer neural networks. In: Ellacott SW, Mason JC and Anderson IJ (eds) Mathematics of Neural Networks, Models, Algorithms and Applications, pp. 250–254. Kluwer Academic Publishers

Michalewicz Z and Fogel DB (2000) How to solve it: Modern Heuristics. Springer

Plagianakos VP and Vrahatis MN (1999) Training Neural Networks with 3-bit Integer Weights. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M and Smith RE (eds) Proceedings of Genetic and Evolutionary Computation Conference (GECCO'99), pp. 910–915. Morgan Kaufmann

Plagianakos VP and Vrahatis MN (2000) Training Neural Networks with Threshold Activation Functions and Constrained Integer Weights. Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN'2000). Como, Italy

Rudolph G (1991) Global optimization by means of distributed evolution strategies. In: Schwefel H-P and Männer R (eds) Parallel problem solving from nature, Lecture Notes in Computer Science, 496, pp. 209–213. Springer, Berlin

Rudolph G (1994) An evolutionary algorithm for integer programming. In: Davidor Y, Schwefel H-P and Männer R (eds) Parallel Problem Solving from Nature, Lecture Notes in Computer Science, 866, pp. 139–148. Springer-Verlag, Berlin

Rumelhart DE, Hinton GE and Williams RJ (1986) Learning internal representations by error propagation. In: Rumelhart DE and McClelland JL (eds) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1, pp. 318–362. MIT Press, Cambridge, Massachusetts

Schwefel H-P (1995) Evolution and Optimum Seeking. John Wiley & Sons, Inc., New York

Storn R (1999) System Design by Constraint Adaptation and Differential Evolution. IEEE Transactions on Evolutionary Computation 3: 22–34

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

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

Yao X and Higuchi T (1999) Promises and Challenges of Evolvable Hardware. Systems, Man, and Cybernetics Part C: Applications and Reviews 29: 87–97