

V. P. Plagianakos · G. D. Magoulas · M. N. Vrahatis

## Evolutionary training of hardware realizable multilayer perceptrons

Received: 5 November 2004 / Accepted: 31 March 2005  
© Springer-Verlag London Limited 2005

**Abstract** The use of multilayer perceptrons (MLP) with threshold functions (binary step function activations) greatly reduces the complexity of the hardware implementation of neural networks, provides tolerance to noise and improves the interpretation of the internal representations. In certain case, such as in learning stationary tasks, it may be sufficient to find appropriate weights for an MLP with threshold activation functions by software simulation and, then, transfer the weight values to the hardware implementation. Efficient training of these networks is a subject of considerable ongoing research. Methods available in the literature mainly focus on two-state (threshold) nodes and try to train the networks by approximating the gradient of the error function and modifying appropriately the gradient descent, or by progressively altering the shape of the activation functions. In this paper, we propose an evolution-motivated approach, which is eminently suitable for networks with threshold functions and compare its performance with four other methods. The proposed evolutionary strategy does not need gradient related information, it is applicable to a situation where threshold activations are used from the beginning of the training, as in “on-chip” training, and is able to train networks with integer weights.

**Keywords** Feedforward neural networks · Backpropagation algorithm · Neural networks with threshold activations · Integer weight neural networks · Integer programming · Steepest descent · Unconstrained optimization · Differential evolution

V. P. Plagianakos (✉) · M. N. Vrahatis  
Computational Intelligence Laboratory, Department  
of Mathematics, University of Patras Artificial Intelligence  
Research Center–UPAIRC, University of Patras,  
GR-26110 Patras, Greece  
E-mail: vpp@math.upatras.gr

G. D. Magoulas  
School of Computer Science and Information Systems,  
University of London, Malet Street, London, WC1E7HX, UK

### 1 Introduction

The multilayer perceptron (MLP) is a widely used neural network model. MLPs consist of many interconnected identical simple processing units, also called neurons. Each unit calculates the inner product of the incoming signals with its weights, adds the bias to the resultant, and passes the calculated sum through its activation function. Units are organized into layers with no feedback connections.

Although units with threshold (binary step) activation functions have been superseded to a large extent by the more computationally powerful units with analogue activations, MLPs with threshold activations are important in that 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 MLPs with sigmoid units and provide a starting point for the study of neural networks properties. Furthermore, when using units with thresholds we can understand the relationship between the size of the network and the complexity of the training [7] better. In [4], it has been demonstrated that MLPs with threshold activations and only one hidden layer, can create any decision region that can be expressed as a finite union of polyhedral sets when there is one unit in the input layer. Moreover, artificially created examples were given where these networks create non convex and disjoint decision regions. Finally, threshold activation functions facilitate and reduce the complexity of neural network implementations in digital hardware and are much less costly to fabricate.

Various modifications of the gradient descent have been presented to train MLPs with threshold activations [2, 5, 6, 12, 25, 28]. However, these methods require to a certain degree, depending on the case, that the learning task is static. Thus, the network is trained “off-line” by applying various problem-dependent heuristics during simulation and, then, the weights are transferred to the

hardware [6]. 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 for “on-chip” training is needed.

In this paper, we propose evolution-motivated strategies that provide the potential advantage of continuing the training process in hardware, when purely threshold activation functions are used.

The paper is organized as follows. In the next sections the training problem of MLPs with threshold activation functions is formulated and current approaches to solve it are discussed. Then, in Sect. 3 an alternative method is described. Section 4 presents experiments and comparative results. Finally, Sect. 5 presents concluding remarks.

## 2 Training networks with threshold activation functions

Consider an MLP with threshold activations consisting of  $L$  layers, in which the first layer denotes the input, the last  $L$  is the output, and the intermediate layers are the hidden layers. It is assumed that the  $(l-1)$  layer has  $N_{l-1}$  units. These units operate according to the following equations:

$$\text{net}_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1} + \theta_j^l, \quad (1)$$

$$y_j^l = \sigma^l(\text{net}_j^l), \quad (2)$$

where  $\text{net}_j^l$  is the network input to the  $j$ -th unit at the  $l$ -th layer,  $w_{ij}^{l-1,l}$  is the connection weight from the  $i$ -th unit at the  $(l-1)$  layer to the  $j$ -th unit at the  $l$ -th layer,  $y_i^{l-1}$  denotes the output of the  $i$ -th unit belonging to the  $l$ -th layer,  $\theta_j^l$  denotes the bias of the  $j$ -th unit at the  $l$ -th layer, and  $\sigma$  is the activation function.

Multilayer perceptrons are usually based on units with analogue activation functions, as the well known sigmoid:

$$s(\text{net}_j^l) = \frac{1}{1 + e^{-\beta \text{net}_j^l}}, \quad (3)$$

where the factor  $\beta$  is introduced to achieve slope modification. For high values of  $\beta$  the sigmoid unit approximates the binary threshold unit [13], i.e.  $\sigma^l(\text{net}_j^l) = \text{“true”}$ , if  $\text{net}_j^l \geq 0$ , and  $\text{“false”}$  otherwise.

Let us now define the error for an output layer unit:  $e_j(t) = d_j(t) - y_j^L(t)$ , for  $j=1,2,\dots,N_L$ , where  $d_j(t)$  is the desired response at the  $j$ -th unit of the output layer for the input pattern  $t$ , and  $y_j^L(t)$  is the output at the  $k$ -th unit of the output layer  $L$ . For a fixed, finite set of input-output cases, the squared error over the training set which contains  $T$  representative cases is:

$$E = \sum_{t=1}^T E(t) = \sum_{t=1}^T \sum_{j=1}^{N_L} e_j^2(t). \quad (4)$$

The fixed-increment rule and the fractional correction rule, both described in [15], were the first training methods for training single layer networks of hard-limiting units. Nowadays, the most common MLP training algorithm, the backpropagation (BP) [21], that incorporates the gradient descent, cannot be applied directly to networks of units with discrete output states, since discrete activation functions (such as binary step function activations) are non-differentiable. Recent research publications have tried to alleviate this problem by considering various modifications of the gradient descent, such as the MRH algorithm [28]. Another training method was proposed by Toms [25], who suggested the use of hybrid activations that are gradually transformed during training from analogue (sigmoid) to thresholds (step functions) depending on the values of a heuristic parameter  $b$ ,  $0 \leq b \leq 1$ . Thus, the hidden unit activations,  $\sigma_h$ , are

$$\sigma_h(\text{net}_j^l) = u(\text{net}_j^l)(1-b) + \left( \frac{2}{1 + e^{-\text{net}_j^l}} - 1 \right) b, \quad (5)$$

where

$$u(\text{net}_j^l) = \begin{cases} 1, & \text{if } \text{net}_j^l \geq 0 \\ -1, & \text{if } \text{net}_j^l < 0 \end{cases}.$$

Note that in Relation (5), when  $b=1$  the hidden units are purely analogue having a sigmoid activation, while for  $b=0$  they become purely discrete with two output states. For intermediate values the network is a hybrid with activation functions that are differentiable everywhere except at  $\text{net}_j^l$ . Note that for the units of the output layer,  $L$ , Toms [25] used the activations

$$\sigma_o(\text{net}_j^L) = \frac{2}{1 + e^{-\text{net}_j^L}} - 1. \quad (6)$$

Bartlett and Downs [2] introduced another approach by defining the weights as random variables with smooth distribution functions and proposed an algorithm that uses an approach that is similar to BP to adjust the parameters of the weights’ distributions. As they point out, their method is similar to BP with regard to computational complexity, but needs additional computations in the estimation of the gradients.

Corwin et al. [6] suggested to train MLPs with progressively steeper analogue functions to facilitate training. In their experiments, they used values such as  $\beta \in \{2, 3, 5, 10\}$  to alter the shape of the sigmoid (c.f. with Eq. (3)) from time to time during training. The mathematical analysis behind their method suggests that this approach is only valid if the network learning error for each particular value of  $\beta$  is “small” [6].

Finally, Goodman [5] proposed an approximation to gradient descent, the so-called pseudo-gradient training method. The pseudo-gradient assumes that units with two discrete output states are used, i.e.  $f$  (or  $-f$ ) for “false” and  $t$  (or  $+t$ ) for “true”, where  $f$ ,  $t$  are real positive numbers and  $f < t$ , instead of the classical 0 and

1 (or  $-1$ , and  $+1$ ). Real positive values prevent units from saturating, give to the logic “false” some power of influence over the next layer of the network, and help the justification of the approximated gradient value.

The idea of the pseudo-gradient was first introduced in training discrete recurrent neural networks [29, 30] and then extended to MLPs with threshold activations [5]. The method approximates the true gradient of the error function with respect to the weights, i.e.  $\nabla E(w)$ , by introducing an analogue set of values for the outputs of the hidden layer units and the output layer units.

Thus, it is assumed that (2) can be written as:

$$y_j^l = \bar{\sigma}^l(s(\text{net}_j^l)), \quad (7)$$

where  $\bar{\sigma}(x) = \text{“true”}$ , if  $x \geq 0.5$ , and  $\text{“false”}$  otherwise, if  $s(\cdot)$  is defined in  $[0, 1]$ . If  $s(\cdot)$  is defined in  $[-1, 1]$ , then  $\bar{\sigma}(x) = \text{“true”}$  if  $x \geq 0$ , and  $\text{“false”}$  otherwise.

Using the chain rule, the pseudo-gradient is computed:

$$\frac{\partial \tilde{E}}{\partial w_{ij}^{l-1,l}} = \tilde{\delta}_j^l y_i^{l-1}, \quad (8)$$

where the approximation of the backpropagating error signal,  $\tilde{\delta}$ , for an output layer unit is

$$\tilde{\delta}_j^L = (d_j - s(\text{net}_j^L)) s'(\text{net}_j^L), \quad (9)$$

and for units of any other layer, i.e.  $l \in [2, L-1]$ , is

$$\tilde{\delta}_j^l = s'(\text{net}_j^l) \sum_n w_{jn}^{l,l+1} \tilde{\delta}_n^{l+1}. \quad (10)$$

In Eqns. (9) and (10), the term  $s'(\text{net}_j^l)$  is the derivative of the analogue activation function.

By using real positive values for “true” and “false” it is ensured that the pseudo-gradient will not reduce to zero when the output is “false”. Note also that the method does not use  $\sigma'$  which is zero everywhere and non-existent at zero. Instead,  $s'$ , which is always positive, is used so that  $\tilde{\delta}_j^l$  gives an indication of the direction and magnitude of a step up or down as a function of  $\text{net}_j^l$  on the surface of the error function  $E$ . The justification of the pseudo-gradient can be found in any one of [5, 29, 30], and is based on the idea of using the gradient of a sigmoid as a heuristic hint instead of the true gradient.

However, as pointed out in [5] the value of the pseudo-gradient is not accurate enough, so gradient descent based training of MLPs with thresholds is considerably slow when compared with BP training of MLPs with continuous activations.

Based on the idea of the pseudo-gradient, in [12] an attractive alternative has been proposed. This method exploits the imprecise information regarding the error function and the approximated gradient, like the pseudo-gradient method does, however it has an improved convergence speed and is potentially useful in situations where the pseudo-gradient method fails to converge.

### 3 Evolution strategies for training MLPs with threshold activations

In this section we introduce a novel approach based on evolution strategies (ESs) for training MLPs with purely threshold units. ESs are adaptive stochastic search methods which mimic the metaphor of natural biological evolution. Distinctly different from other adaptive stochastic search algorithms, evolutionary computation techniques operate on a set of potential solutions, which is called *population*, applying the principle of survival of the fittest to produce better and better approximations to a solution, and, through cooperation and competition among the potential solutions, they find the optimal one. This approach often helps finding optima in complicated optimization problems more quickly than traditional optimization methods. The main differences between ESs and genetic algorithms (GAs) lie in that the mutation operator is the key self-adaptation feature of the ESs, while GAs prefer smaller mutation probability (rate) [1, 22]. A high level description of a general ES is presented here.

Evolutionary strategy model

```

{
//initialise the time counter
t := 0;
//initialise the population
InitPopulation(P(t));
//evaluate fitness of all individuals
F_P(t) := Evaluate(P(t));
//test for termination criterion
while not done do
t := t + 1;
//select sub-population
Q(t) := SelectParents(P(t));
//recombine the “genes”
R(t) := Recombine(Q(t));
//perturb the mated population
M(t) := Mutate(R(t));
//evaluate the new fitness
F_M(t) := Evaluate(M(t));
//select the survivors
P(t + 1) := Survive(F_P(t), F_M(t));
end
}

```

Here, we use the differential evolution (DE) strategies, which have been designed as stochastic parallel direct search methods that can efficiently handle non differentiable, nonlinear and multimodal objective functions, and require few, easily chosen control parameters [23]. Experimental results have shown that DE algorithms have good convergence properties, outperform other classical or evolutionary methods [16, 17, 23], are able to train MLPs with integer weights [18, 19] and can be efficiently implemented in parallel [20].

To apply DE algorithms for training MLPs with thresholds, we start with a fixed number ( $NP$ ) of  $N$ -dimensional weight vectors, as an initial weight population, and evolve them over time. The number of individuals,  $NP$ , is kept fixed throughout the learning

process and the population is initialized randomly following a uniform probability distribution. As in ESs, at each iteration of the DE algorithm, called generation, new weight vectors are generated by the combination of randomly chosen weight vectors from the current population, using one of the following relations:

$$w_{g+1}^j = w_g^{r_1} + \mu(w_g^{r_2} - w_g^{r_3}), \quad (11)$$

or

$$w_{g+1}^j = w_g^j + \mu(w_g^{\text{best}} - w_g^j) + \mu(w_g^{r_1} - w_g^{r_2}), \quad (12)$$

where  $w_g^{\text{best}}$  is the best member of the previous generation,  $\mu > 0$  is a real parameter, called mutation constant, which regulates the contribution of the difference between two weight vectors, and

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

are random integers mutually different and different from the running index  $i$ .

The outgoing weight vectors are then mixed with another predetermined weight vector, the target weight vector. This operation, which is called crossover, is used to further increase the diversity of the mutant weight vector. Specifically, for each component  $j$  of the mutant weight vector, we randomly choose a real number  $r$  in the interval  $[0,1]$ . Then, this number is compared with the crossover constant  $\rho$ ; if  $r \leq \rho$ , we replace the  $j$ -th component of the trial vector with the  $j$ -th component of the mutant vector; otherwise, we pick the  $j$ -th component of the target vector.

The crossover operator yields the so-called trial weight vector that 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. In algorithm 1, we illustrate a high level description of the DE training algorithm.

## 4 Applications

In this section, we present four sets of experiments applying the DE algorithms of relation (11), named  $DE_1$ , and of relation (12), named  $DE_2$ , and the algorithms proposed in [5, 6, 12, 25], which are denoted in the tables below as GZ, GLO, MVGA and T, respectively.

---

### Algorithm 1: Differential evolution

---

Initialize the population of individuals

Evaluate fitness of all individuals

#### Repeat

For  $i = 1$  to  $NP$

MUTATION( $w_g^i$ )  $\rightarrow$  Mutant\_Vector

CROSSOVER(Mutant\_Vector)  $\rightarrow$  Trial\_Vector

If  $E(\text{Trial\_Vector}) \leq E(w_g^i)$ ,

accept Trial\_Vector for the next generation

EndFor

Until the termination criterion is met

---

For the DE algorithms, no effort has been made to tune the mutation and crossover constants,  $\mu$  and  $\rho$  respectively. We have used the fixed values  $\mu = 0.5$  and  $\rho = 0.7$ , instead.

One thousand simulation runs have been made in each case and the average performance results reported below have been validated in all cases using the well-known  $t$ -test for statistical significance at the level  $\alpha \leq 0.05$  (see, for example, [11]), using the SPSS 13 statistical software package. To this end, we have conducted 1000 more independent runs for each algorithm. The difference between the original values and the values computed using the new simulations are not significantly different, at the significance level  $\alpha \leq 0.05$ , i.e. the original value is equal to the new one with probability  $(1-\alpha) = 95\%$ , when those values are indeed equal.

### 4.1 Training by gradually altering the sigmoid activations

In this set of experiments, we train MLPs with threshold activation functions by gradually transforming the sigmoid activations to thresholds. This approach is adopted by both the GLO and T algorithms and is useful for learning tasks that do not change over time. In this case, it may be sufficient to determine the weights by means of software simulation and, at a later time, transfer these values to the hardware implementation. Note that this approach is only applicable to train MLPs with threshold activation functions for which the weights are not required to be updated in hardware. More specifically, for the GLO and the T algorithms the activations were altered according to the guidelines given in [6] and in [25].

To test the efficiency of the proposed evolutionary strategies, the  $DE_1$ , and  $DE_2$  algorithms were tested under the same conditions. When the input patterns were correctly classified and the network error was small, the value of the sigmoid gain,  $\beta$ , was altered in the sequence (1, 10, 20, 30, 40, 50,  $\infty$ ), in order to increase the slope of the analogue activation function (sigmoid).

#### 4.1.1 The XOR classification problem [2, 5, 6]

Classification of the four XOR patterns in one of two classes,  $\{0, 1\}$ , is sensitive to initial weights, presents a multitude of local minima, and it is known to have solutions for threshold units [21]. We used the 2-2-1 classic architecture and set the learning rate to 0.1. In all in-

**Table 1** Results for the XOR Problem

| Algorithm | Min | Mean  | Max   | SD    | Succ. (%) |
|-----------|-----|-------|-------|-------|-----------|
| $DE_1$    | 270 | 432.4 | 1,062 | 106.6 | 100       |
| $DE_2$    | 252 | 394.0 | 1,170 | 111.9 | 100       |
| GLO       | 142 | 335.6 | 2,303 | 250.4 | 84        |
| T         | 117 | 192.2 | 528   | 66.4  | 69        |

stances, 1,000 simulations were run and the results are summarized in Table 1, where the following notation is used: Mean indicates the mean number of function evaluations; SD the standard deviation of function evaluations; Max the maximum number of function evaluations; Min the minimum number of function evaluations and Succ. the percentage of successful runs.

When interpreting the results shown in Table 1, the reader should keep in mind that the GLO and the T algorithms use at each iteration one gradient and one error function evaluation to update the weights, while the DE algorithms do not require gradient evaluations. This means that in learning the XOR, the algorithms GLO and T required, in addition to the error function evaluations shown in Table 1, an average of 335.6 and 192.2 gradient evaluations, respectively. Note that a gradient evaluation sometimes is considered as having the cost of three error function evaluations [14].

#### 4.1.2 The 3-bit parity problem [8]

A 3-3-1 MLP receives eight three-dimensional binary input patterns and must output an “1” if the inputs have an odd number of ones and “0” if the inputs have an even number of ones. Geometrically, the input space for the 3-parity problem is a cube in a three-dimensional space. This is a very difficult problem for an MLP because the network must determine the proper parity (the value at the output) for input patterns which differ only by Hamming distance 1. The stepsize was equal to 0.5. The results of 1,000 simulations are summarized in Table 2.

From the results of Table 2, it is clear that the GLO algorithm outperforms all other methods tested. The success performance of the DE algorithms was significantly high (100%), but the average number of error function evaluations was high as well. This is due to the population-based nature of the method.

#### 4.2 Training by explicitly using threshold activation functions

Here, we further expand our experiments to train “on-chip” MLPs by explicitly using threshold activation functions. This approach provides the advantage of training or continuing the training process on hardware, when purely threshold activation functions are used.

We have compared the GZ and MVGA algorithms, which are able to train MLPs having only threshold

activations, against the DE<sub>2</sub> algorithm which gave the best performance in the first set of experiments. The other algorithms tested in the first set of experiments, i.e. the GLO and the T, are not able to train an MLP when threshold activations are used from the beginning of the training.

The DE<sub>2</sub> trained the MLP using threshold activation functions and 5-bit integer weights. The use of integer weights reduces the amount of memory required for weight storage in hardware implementations. Additionally, it simplifies the digital multiplication operation, since multiplying any number with a  $k$ -bit integer requires only the following number of basic instructions: one sign change,  $(k-1)(k-2)/2$  one-step left shifts and  $(k-2)$  additions. Finally, if inputs are restricted to the set  $\{-1, 1\}$  (bipolar inputs), the units in the first hidden layer require only sign changes during multiplication operations, and only integer additions.

Note that the DE<sub>2</sub> needs only the values of the error function to update the weights, while the GZ and MVGA algorithms are both based on the idea of pseudo-gradient to calculate the weight corrections and use real numbers for the weights [5, 12].

In Table 3, results for the XOR problem are shown. The performance of the DE<sub>2</sub> is significantly high. The DE<sub>2</sub> reveals the highest percentage of success in the experiments, i.e. 95 successful simulation runs out of 1,000. The MVGA algorithm has better performance than the original pseudo-gradient method, GZ. However, when compared to the DE<sub>2</sub>, the MVGA requires, an average number of 280.6 gradient evaluations in addition to the error function evaluations shown in Table 3.

In Table 4, results from the 3-bit parity problem are exhibited. The DE<sub>2</sub>, needing an average of 1,273.1 error function evaluations and no gradient calculations, converges in 880 out of the 1,000 simulations runs. The modified pseudo-gradient algorithm, MVGA, trains the network, needing on the average 702.5 error function and 702.5 pseudo-gradient evaluations, while the original method, GZ, is not able to learn the training set.

**Table 2** Results for the 3-bit parity problem

| Algorithm       | Min | Mean    | Max   | SD      | Succ. (%) |
|-----------------|-----|---------|-------|---------|-----------|
| DE <sub>1</sub> | 672 | 1,732.8 | 7,488 | 825.6   | 100       |
| DE <sub>2</sub> | 640 | 1,423.8 | 9,888 | 1,195.8 | 100       |
| GLO             | 77  | 1,46.3  | 372   | 50.8    | 96        |
| T               | 70  | 114.9   | 448   | 45.3    | 88        |

**Table 3** Results for the XOR problem using threshold activations

| Algorithm       | Min   | Mean    | Max   | SD      | Succ. (%) |
|-----------------|-------|---------|-------|---------|-----------|
| DE <sub>2</sub> | 36    | 368.3   | 1,692 | 345.6   | 95        |
| GZ              | 5,202 | 5,202.7 | 9,964 | 3,530.8 | 5         |
| MVGA            | 122   | 280.6   | 334   | 56.1    | 50        |

**Table 4** Results for the 3-bit parity problem using threshold activations

| Algorithm       | Min | Mean    | Max   | SD    | Succ. (%) |
|-----------------|-----|---------|-------|-------|-----------|
| DE <sub>2</sub> | 64  | 1,273.1 | 3,072 | 750.9 | 88        |
| GZ              | –   | –       | –     | –     | 0         |
| MVGA            | 320 | 702.5   | 2,964 | 833.3 | 55        |

**Table 5** 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       |
| DE <sub>1</sub> | 100        | 100        | 100        |
| DE <sub>2</sub> | 100        | 100        | 100        |

### 4.3 MONK’s problems and generalization results

In addition to training speed and efficiency, we have also evaluated the generalization performance of the DE algorithms. To this end, we have also tested them on the MONK’s problems [24]. These three problems from the UCI Machine Learning Repository [3] are difficult binary classification tasks which have been used for comparing the generalization performance of learning algorithms. They rely on the artificial robot domain, in which robots are described by six different attributes. 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 [24], for details). Each possible value for every attribute is assigned a single bipolar input, resulting 17 inputs.

We have compared the DE<sub>1</sub> and DE<sub>2</sub> algorithms utilizing threshold functions and 5-bit integer weights against the well known backpropagation (BP), the backpropagation with weight decay (BPWD), and the cascade correlation (CC) algorithms, using the experimental settings found in [24]. The termination condition was a training error less than 0.1 and the maximum allowed iterations were 5,000. The BP, the BPWD and the CC training algorithms use sigmoid activation functions and compute the gradient of the error function at each iteration. In Table 6 we exhibit the best generalization accuracy results of the algorithms on the MONK’s problems.

It is clear from Table 5 that the DE algorithms generate MLPs, which are at least as capable as the best generated by real-weight learning algorithms using sigmoid activations. 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.

The topology of the trained networks is shown in Table 6. It is known that the best generalizers are neither too complex nor too simple; they exactly match the complexity of the concept which is embedded in the training data. We think that the reason why the proposed algorithms, in general, need a larger network to generate MLPs with good generalization capabilities is that more integers than real numbers are needed to match the complexity of the given problem [9].

**Table 6** 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 |
| DE <sub>1</sub> | 17:4:1 | 17:4:1 | 17:3:1 |
| DE <sub>2</sub> | 17:4:1 | 17:4:1 | 17:3:1 |

### 4.4 Training a neural controller

In this last experiment, an MLP is trained to control a real-life mechatronic device. A cutting tool, driven by a servo-motor, is augmented with a force sensor which returns a signal contaminated by tool chatter and unwanted noise to the controller [26, 27].

A 2-4-1 neural controller tries to maintain a constant force on the tool by varying the material feed rate. The controller generates a signal to the actuator to get the necessary optimum feed rate, and, in this way, to achieve the desired product quality. Feed rate demand is used as the input to the device, and the cutting force, as measured by the force sensor on the workpiece, is the device output. The neural controller is trained using fuzzified values for the control system error between desired and actual force  $e_k$ , and the error change  $De_k = e_k - e_{k-1}$ . The controller provides a correction signal  $Du_k$  (desired output) which passes through an integrator to give the control input to the device [10, 27].

The training set consists of thirteen input-output samples, i.e.  $\{input, output\} = \{(e_k, De_k), Du_k\}$ , where all variables take values in the normalized space  $[-1, +1]$ . These samples correspond to 13 out of 49 possible linguistic rules to control the lathe cutting process; the rest of the rules are used for testing [10]. The encoding of the linguistic rules and the dataset are described in detail in (p 186, [10]). The termination condition was a training error less than 0.1 and the maximum allowed iterations were 20,000.

Threshold activations and 5-bit integer weights were used when training with the DE<sub>2</sub> algorithm; real numbers for the weights were used for all other methods. For the GLO and T algorithms the activations were altered according to the guidelines given in [6] and in [25].

In Table 7, we exhibit results for this experiment using the five methods in 1,000 simulation runs. Only DE<sub>2</sub> and MVGA managed to train the neuro-controller. All other methods failed in all cases. The DE<sub>2</sub> algorithm is able to train with significant success and provides the

**Table 7** Results from training the MLP neuro-controller

| Algorithm       | Min   | Mean    | Max    | SD      | Succ. (%) |
|-----------------|-------|---------|--------|---------|-----------|
| DE <sub>2</sub> | 1,020 | 4,400.6 | 15,390 | 2,716.2 | 93        |
| GLO             | –     | –       | –      | –       | 0         |
| T               | –     | –       | –      | –       | 0         |
| GZ              | –     | –       | –      | –       | 0         |
| MVGA            | 1,200 | 2,666.1 | 11,114 | 917.2   | 34        |

advantage of retraining the neuro-controller using threshold activations. Note that retraining is especially useful for real-life learning problems and tasks that slowly change over time.

---

## 5 Conclusions

Various methods for training networks with threshold activations have been proposed in the literature. However, these methods exhibit slow convergence speed and low percentage of success compared to networks with continuous activations. It is known that the hardware implementation of the backward passes, which compute the gradient of the error function, is more difficult than the implementation of the forward passes. Thus, in this paper, we proposed evolutionary strategies that do not perform gradient evaluations and are able to train MLPs with purely threshold activations. The behavior of the algorithms has been examined by means of experiments and comparative results have been presented. The performance of the algorithms is promising even when compared with other methods that require the gradient approximations of the error function, or train the network by progressively altering the shape of the activation functions.

Furthermore, the proposed strategies were tested for training MLPs with threshold activations and integer weights. MLPs of this type constitute an interesting alternative class of neural networks, as they require significantly less amount of memory for the storage of their weights and uncomplicated digital arithmetic operations when compared to networks with real weights and non-linear (sigmoid) activation functions.

In a future communication we intend to present the behavior and characteristics of the DE algorithms in training large MLPs with thresholds and big data sets. Preliminary results are promising, but further work is needed to optimize the DE algorithms' performance in such tasks.

**Acknowledgements** The authors would like to thank the European Social Fund, Operational Program for Educational and Vocational Training II (EPEAEK II), and particularly the Program PYTHAGORAS for funding the above work. Dr V.P. Plagianakos and Prof. M.N. Vrahatis acknowledge the financial support of the University of Patras Research Committee through a "Karatheodoris" research grant. We also acknowledge the help of Prof. R.E. King of the Department of Electrical and Computer Engineering at the University of Patras, Greece, in the neuro-controller training experiment. The authors wish to thank the Editor and the referees for constructive comments and useful suggestions.

---

## References

1. Bäck T, Schwefel HP (1993) An overview of evolutionary algorithms for parameter optimization. *Evol Comput* 1:1–23
2. Bartlett PL, Downs T (1992) Using random weights to train multilayer networks of hard-limiting units. *IEEE Trans Neural Netw* 3:202–210
3. Blake CL, Merz CJ (2005) UCI Repository of machine learning databases. University of California, Irvine, Department of Information and Computer Sciences, URL: <http://www.ics.uci.edu/~mllearn/MLRepository.html>, last accessed 4/2005
4. Gibson GJ, Cowan FN (1990) On the decision regions of multilayer perceptrons. *Proc IEEE* 78:1590–1594
5. Goodman R, Zeng Z (1994) A learning algorithm for multilayer perceptrons with hard-limiting threshold units. In: *Proceedings of the IEEE Neural Networks for Signal Processing*, pp 219–228
6. Gorwin EM, Logar AM, Oldham WJB (1994) An iterative method for training multilayer networks with threshold functions. *IEEE Trans Neural Netw* 5:507–508
7. Hampson SE, Volper DJ (1990) Representing and learning boolean functions of multivalued features. *IEEE Trans Syst Man Cybern* 20:67–80
8. Hohil ME, Liu D, Smith SH (1999) Solving the N-bit parity problem using neural networks. *Neural Networks* 12:1321–1323
9. Khan AH (1996) Feedforward neural networks with constrained weights. PhD Thesis, University of Warwick, Department of Engineering
10. King RE (1999) *Computational Intelligence in Control Engineering*. Marcel Dekker Inc., New York
11. Law AM, Kelton WD (2000) *Simulation modeling and analysis*, 3rd edn. McGraw-Hill, New York
12. Magoulas GD, Vrahatis MN, Grapsa TN, Androurakis GS (1997) A training method for discrete multilayer neural networks. In: Ellacot SW, Mason JC, Anderson IJ (eds) *Mathematics of neural networks: models, algorithms & applications*, chapter 41. Kluwer, Operations Research/Computer Science Interfaces series
13. McCullough W, Pitts WH (1943) A logical calculus of the ideas imminent in nervous activity. *Bull Math Biophys* 5:115–133
14. Møller MF (1993) A scaled conjugate gradient algorithm, for fast supervised learning. *Neural Networks* 6:525–533
15. Nilsson NJ (1965) *Learning Machines*. McGraw-Hill, New York
16. Plagianakos VP, Magoulas GD, Nouis NK, Vrahatis MN (2001) Training multilayer networks with discrete activation functions. In: *Proceedings of the INNS–IEEE international joint conference on neural networks (IJCNN2001)*, Washington DC, USA, pp 2805–2810
17. Plagianakos VP, Sotiropoulos DG, Vrahatis MN (1998) Integer weight training by differential evolution algorithms. In: Mastorakis NE (ed) *recent advances in circuits and systems*. World Scientific pp 327–331
18. Plagianakos VP, Vrahatis MN (1999) Training neural networks with 3-bit integer weights. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) *Proceedings of genetic and evolutionary computation conference (GECCO'99)*. Morgan Kaufmann, Orlando, pp 910–915
19. Plagianakos VP, Vrahatis MN (1999) Neural network training with constrained integer weights. In: Angeline PJ, Michalewicz Z, Schoenauer M, Yao X, Zalzala A (eds) *Proceedings of congress on evolutionary computation (CEC'99)*. IEEE Press, Washington DC, pp 2007–2013
20. Plagianakos VP, Vrahatis MN (2002) Parallel evolutionary training algorithms for 'hardware-friendly' neural networks. *Natural Computing* 1:307–322
21. Rumelhart DE, McClelland JL (eds) *Parallel distributed processing: explorations in the microstructure of cognition*. MIT Press, New York 1:318–362
22. Srinivas M, Patnaik L (1994) Genetic algorithms: a survey. *IEEE Computer*, pp 17–26
23. Storn R, Price K (1997) Differential Evolution—A simple and efficient heuristic for global optimization over continuous spaces. *J Global Optim* 11:341–359
24. 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, Zhang J (1991) The MONK's problems: a performance comparison of different learning algorithms. Technical Report, Carnegie Mellon University, CMU-CS-91-197
25. Toms DJ (1990) Training binary node feed forward neural networks by back-propagation of error. *Electron Lett* 26:1745–1746
  26. Tomizuka M, Zhang S (1988) Modeling and conventional/adaptive PI control of a lathe cutting process. *Trans ASME* 110:305–354
  27. Tsitouras G, King R (1997) Rule-based neural control of mechatronic systems. *Int J Intelligent Mechatronics* 2:1–11
  28. Widrow B, Winter R (1988) Neural nets for adaptive filtering and adaptive pattern recognition. *IEEE Computer*, March, 25–39
  29. Zeng Z, Goodman R, Smyth P (1993) Learning finite state machines with self-clustering recurrent networks. *Neural Comput* 5:976–990
  30. Zeng Z, Goodman R, Smyth P (1994) Discrete recurrent neural networks for grammatical inference. *IEEE Trans Neural Networks* 5:320–330