Hardware-Friendly Higher-Order Neural Network Training Using Distributed Evolutionary Algorithms

M.G. Epitropakis[a], V.P. Plagianakos[b], M.N. Vrahatis[*,a]

[a]*Computational Intelligence Laboratory (CI Lab), Department of Mathematics, University of Patras, GR–26110 Patras, Greece.*
[b]*Department of Computer Science and Biomedical Informatics, University of Central Greece, Papassiopoulou 2-4, GR–35100 Lamia, Greece.*

Abstract

In this paper, we study the class of Higher-Order Neural Networks and especially the Pi-Sigma Networks. The performance of Pi-Sigma Networks is evaluated through several well known neural network training benchmarks. In the experiments reported here, Distributed Evolutionary Algorithms are implemented for Pi-Sigma neural networks training. More specifically the distributed versions of the Differential Evolution and the Particle Swarm Optimization algorithms have been employed. 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 approach is applied to train Pi-Sigma networks using threshold activation functions. Moreover, the weights and biases were confined to a narrow band of integers, constrained in the range $[-32, 32]$. Thus, the trained Pi-Sigma neural networks can be represented by using 6 bits. Such networks are better suited than the real weight ones for hardware implementation and to some extend are immune to low amplitude noise that possibly contaminates the training data. Experimental results suggest that the proposed training process is fast, stable and reliable and the distributed trained Pi-Sigma networks exhibited good generalization capabilities.

[*]Corresponding author: Phone: +30 2610 997374, Fax: +30 2610 992965
Email addresses: `mikeagn@math.upatras.gr` (M.G. Epitropakis), `vpp@math.upatras.gr` (V.P. Plagianakos), `vrahatis@math.upatras.gr` (M.N. Vrahatis)

Key words: Pi-Sigma Networks, Distributed Differential Evolution, Distributed Particle Swarm Optimization, Backpropagation Neural Networks, Integer Weight Neural Networks, Threshold Activation Functions, "Hardware-Friendly" Implementations, 'On-chip' Training, Higher-Order Neural Networks
2000 MSC: 65K10, 68T20, 68W10, 68W15, 90C10, 90C56, 78M50, 62M45

1. Introduction

Evolutionary Algorithms (EAs) are nature inspired problem solving optimization algorithms, which employ computational models of evolutionary processes. Various evolutionary algorithms have been proposed in the literature. The most important ones are: Genetic Algorithms [1, 2], Evolutionary Programming [3, 4], Evolution Strategies [5, 6], Genetic Programming [7], Particle Swarm Optimization [8] and Differential Evolution algorithms [9]. The algorithms mentioned above share the common conceptual base of simulating the evolution of a population of individuals using a predefined set of operators. Generally, the operators utilized belong to one of the following categories: the *selection* and the *search* operators. The most commonly used search operators are *mutation* and *recombination*. EA's are parallel and distributed implementations and they are inspired by niche formation. Niche formation is a common biological phenomenon [10]. Niches could aid the differentiation of the species by imposing reproduction restrictions. Many natural environments can lead to niche formation. For example, remote islands, high mountains and isolated valleys, restrict the species and therefore the evolution process. Although diversity tends to be low in each subpopulation, overall population diversity is maintained through isolation. However, occasionally an individual may escape and reach nearby niches, increasing the diversity of their populations [10].

In this paper, we study the class of Higher-Order Neural Networks (HONNs) and in particular Pi-Sigma Networks (PSNs), which were introduced by Shin and Ghosh [11]. Although PSNs employ fewer weights and processing units than HONNs they manage to indirectly incorporate many of their capabilities and strengths. PSNs have effectively addressed several difficult tasks, where traditional Feedforward Neural Networks (FNNs) are having difficulties, such as zeroing polynomials [12] and polynomial factorization [13]. Here, we study PSN's performance on several well known neural network training problems. In our experiments, we trained PSNs with small integer weights and threshold activation functions, utilizing distributed Evolutionary Algorithms. More specifically, modified distributed versions of the Differential Evolution (DE) [9, 14] and Particle Swarm Optimization (PSO) [8, 15] algorithms have been used. DE and PSO have proved to be effective and efficient optimization methods on numerous hard real-life problems [14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. The distributed EAs has been designed keeping in mind that the resulting integer weights and biases require less bits to be stored and the digital arithmetic operations between them are easier to be implemented in hardware. An additional advantage of the proposed approach is that no gradient information is required; thus (in contrast to classical methods) no backward passes were performed.

Hardware implemented PSNs with integer weights and threshold activation functions can continue training, even during the operation of the system, if the input data are changing (on-chip training) [14, 19]. Another advantage of neural networks with integer weights and threshold activation functions is that the trained neural networks are to some extend immune to noise in the training data. Such networks only capture the main feature of the dataset. 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 [14].

If the network is trained in a constrained weight space, smaller weights are found and less memory is required. On the other hand, the network training procedure can be more effective and efficient when larger integer weights are allowed. Thus, for a given application a trade off between effectiveness and memory consumption has to be considered. Here, Pi-Sigma neural networks with six bit weight representation have been utilized, i.e. integer weights confined in the range $[-32, 32]$. Although the weights are restricted, the trained PSNs can effectively tackle several benchmark problems, as presented in the experimental results.

The remaining of this paper is organized as follows. Section 2 reviews various parallel Evolutionary Algorithm implementations. Section 3 briefly describes the mathematical model of HONNs and PSNs. Section 4 is devoted to the presentation of the distributed DE and PSO optimization algorithms. Extensive experimental results are presented in Section 5. The paper ends with a discussion and concluding remarks.

2. Parallel and Distributed Evolutionary Algorithms

Following the biological niche formation many parallel and distributed Evolutionary Algorithm implementations exist [24, 25, 26, 27, 28, 29]. The most widely known are [24, 25, 29]:

a) single-population (global) master-slave algorithms,

b) single-population fine-grained algorithms,

c) multiple-population coarse-grained algorithms, and

d) hierarchical parallel algorithms (hybrid approach).

In EA literature *single-population fine-grained algorithms* are also called *cellular EAs* (cEAs). The *multiple-population coarse-grained algorithms* are also known as *island models* or *distributed EAs* (dEAs). These two approaches are most popular among EA researchers and seem to provide a better sampling of the search space. Additionally, they improve the numerical and runtime behavior of the basic algorithm [10, 24, 25, 29].

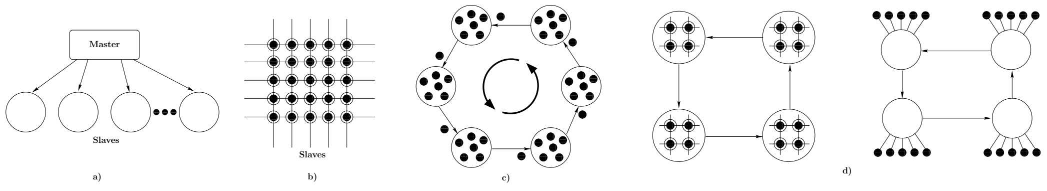


Figure 1: Parallel and distributed Evolutionary Algorithms: a) single-population master-slave algorithms, b) single-population fine-grained algorithms, c) multiple-population coarse-grained algorithms, and d) hybrid approaches

In a *master-slave* implementation there exists a single panmictic population (selection takes place globally and any individual can potentially mate with any other), but the evaluation of the fitness of each individual is performed in parallel among many processors. This approach does not affect the behavior of the EA algorithm; the execution is identical to a basic sequential EA.

According to the cEA approach each individual is assigned to a single processor and the selection and reproduction operators are limited to a small local neighborhood. Neighborhood overlap is permitting some interaction among all the individuals and allows a smooth diffusion of good solutions across the population.

We must note that one could use a uniprocessor machine to run cEAs and dEAs and still get better results than with sequential panmictic EAs. The main difference between cEAs and dEAs is the separation of individuals into distinct subpopulations (islands). In biological terms, dEAs resembles distinct semi-isolated populations in which evolution takes place independently. dEAs are more sophisticated as they occasionally exchange individuals between subpopulations, utilizing the migration operator. The migration operator defines the topology, the migration rate, the migration interval, and the migration policy [25, 26, 30, 31]. The migration topology determines island interconnections. The migration rate is the number of individuals exchanged during the migration. The migration interval is the number of generations between two consecutive calls of the operator, while the migration policy defines the exchanged individuals and their integration within the target subpopulations. The migration rate and migration interval are the two most important parameters, controlling the quantitative aspects of migration [24, 25]. In the case where the genetic material, as well as the selection and recombination operators, are the same for all the individuals and all subpopulations of a dEA, we call these algorithms *uniform*. On the other hand, when different subpopulations evolve with different parameters and/or with different individual representations, the resulting algorithm is called *nonuniform* dEA [32, 33]. For the rest of the paper we focus on uniform dEAs.

Hierarchical parallel algorithms combine at least two different methods of EA parallelization to form a hybrid algorithm. At the higher level exists a multiple-population EA algorithm, while at the lower levels any kind of parallel EA implementation can be utilized.

Algorithm 1 exhibits in pseudocode the asynchronous island model of an EA (uniform dEA), which is executed in n parallel or distributed processors. Both Distributed Differential Evolution and Distributed Particle Swarm Optimization algorithms are based on this scheme.

To conclude, the use of parallel and distributed EA implementation has many advantages [32], such as:

1. finding alternative solutions to the same problem,
2. parallel search from multiple points in the space,
3. easy parallelization,

Algorithm 1 The asynchronous island model

```
1: Initialize the population
2: while termination criteria are not satisfied do
3:    Perform an EA step
4:    if the solution is found then
5:       Broadcast a termination message
6:    end if
7:    if migration interval is satisfied then
8:       Select and send $n$ individuals according to the migration rate, topology and policy
9:    end if
10:   while the receive buffer is not empty do
11:      Receive migrants according to migration policy
12:   end while
13: end while
```

4. more efficient search, even without parallel hardware,
5. higher efficiency than sequential EAs, and
6. speedup due to the use of multiple CPUs.

For more information regarding parallel EA implementations, software tools, and theory advances the interested reader could refer to the following review papers and books [25, 26, 32, 34, 35].

3. Higher-Order Neural Networks and Pi-Sigma Networks

Higher-order Neural Networks (HONNs) expand the capabilities of standard Feedforward Neural Networks (FNNs) by including input nodes which provide the network with a more complete understanding of the input patterns and their relations. Basically, the inputs are transformed so that the network does not have to learn the most basic mathematical functions, such as squares, cubes, or sines. The inclusion of these functions enhances the network's understanding of a given problem and has been shown to accelerate training on some applications. However, typically only second order networks are considered in practice. The main disadvantage of HONNs is that the required number of weights increases exponentially with the dimensionality of the input patterns.

On the other hand, a Pi–Sigma Network (PSN) utilizes product (instead of summation) nodes as the output units to indirectly incorporate some of the capabilities of HONNs, while using fewer weights and processing units. Specifically, PSN is a multilayer feedforward network that outputs products of sums of the input components. It consists of an input layer, a single 'hidden' (or middle) layer of summing units, and an output layer of product units. The weights connecting the input neurons to the neurons of the middle layer are adapted during the learning process by the training algorithm, while those connecting the neurons of the middle layer to the product units of the output layer are fixed. For this reason the middle layer is not actually hidden and the training process is significantly simplified and accelerated [11, 36, 37].

Let the input $x = (1, x_1, x_2, \ldots, x_N)^\top$, be an $(N+1)$-dimensional vector, where 1 is the input of the bias unit and $x_k, k = 1, 2, \ldots, N$ denotes the k-th component of the input vector. Each neuron in the middle layer computes the sum of the products of each input with the corresponding weight. Thus, the output of the j-th neuron in the middle layer is given by the sum:

$$h_j = w_j^\top x = \sum_{k=1}^{N} w_{kj} x_k + w_{0j},$$

where $j = 1, 2, \ldots, K$ and w_{0j} denotes a bias term. Output neurons compute the product of the aforementioned sums and apply an activation function on this product. An output neuron returns:

$$y = \sigma\left(\prod_{j=1}^{K} h_j\right),$$

where $\sigma(\cdot)$ denotes the activation function. The number of neurons in the middle layer defines the order of the PSN. This type of networks are based on the idea that the input of a K-th order processing unit can be represented by a product of K linear combinations of the input components. Assuming that $(N+1)$ weights are associated with each summing unit, there is a total of $(N+1)K$ weights and biases for each output unit. If multiple outputs are required (for example, in a classification problem), an independent summing layer is required for each one. Thus, for an M-dimensional output vector y, a total of $\sum_{i=1}^{M}(N+1)K_i$ adjustable weight connections are needed, where K_i is the number of summing units for the i-th output. This allows great flexibility as the output layer indirectly incorporates some of the capabilities of HONNs utilizing a smaller number of weights and processing units. Furthermore, the network can be either regular or can be easily incrementally expandable, since the order of the network can be increased by adding another summing unit in the middle layer without disturbing the already established connections.

A further advantage of PSNs is that we do not have to pre-compute the higher order terms and incorporate them into the network, as is necessary for a single layer HONN. PSNs are able to learn in a stable manner even with fairly large learning rates [11, 36, 37]. The use of linear summing units makes the convergence analysis of the learning rules for PSN more accurate and tractable. The price to be paid is that the PSNs are not universal approximators. Despite that, PSNs demonstrated competent ability to solve many scientific and engineering problems, such image compression [38], and pattern recognition [11].

Although FNNs and HONNs can be simulated in software, hardware implementation is required in real life applications, where high speed of execution is necessary. Thus, the natural implementation of FNNs or HONNs (because of their modularity) is a distributed (or parallel) one [14]. In the next section we present the distributed EA used in this study.

4. Neural Network Training Using Distributed Evolutionary Algorithms

For completeness purposes let us briefly present the distributed versions of Differential Evolution and Particle Swarm Optimization algorithms for higher order neural network training. Our distributed implementations are based on the Message Passing Interface standard, which facilitates the execution of parallel applications.

4.1. The Distributed DE Algorithm

Differential Evolution (DE) is an optimization method, that utilizes concepts borrowed from the broad class of Evolutionary Algorithms. DE is capable of handling non-differentiable, discontinuous and multimodal objective functions. The method requires few, easily chosen, control parameters. Extensive experimental results have shown that DE has good convergence properties and in many cases outperforms other well known evolutionary algorithms. The original DE algorithm as well as its distributed implementation have been successfully applied to FNN training [14, 16, 20]. Distributed Differential Evolution (DDE) for Pi-Sigma networks training is presented here.

More specifically, the modified DDE algorithm is a uniform dEA. DDE maintains distinct subpopulations (islands) of potential integer solutions, *individuals*, to probe the search space. Each subpopulation of individuals is randomly initialized in the optimization domain. At each iteration, called *generation*, new individuals are generated through the combination of randomly chosen individuals of the current subpopulation. Starting with a subpopulation of NP integer weight vectors, $w_g^i, i = 1, 2, \ldots, NP$, where g denotes the current generation, each weight vector undergoes mutation to yield a mutant vector, u_{g+1}^i. The mutant vector that is considered here (for alternatives see [9, 39]), is obtained through one of the following equations:

$$\begin{aligned} u_{g+1}^i &= w_g^{\text{best}} + F(w_g^{r_1} - w_g^{r_2}), & (1) \\ u_{g+1}^i &= w_g^{r_1} + F(w_g^{r_2} - w_g^{r_3}), & (2) \end{aligned}$$

where w_g^{best} denotes the best member of the current generation and $F > 0$ is a real parameter, called *mutation constant* that controls the amplification of the difference between the two weight vectors. Moreover, $r_1, r_2, r_3 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\}$ 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 subpopulation 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 hypercube $[-32, 32]^N$, we calculate u_{g+1}^i using the following formula:

$$u_{g+1}^i = \operatorname{sign}(u_{g+1}^i) \times \left(\left|u_{g+1}^i\right| \mod 32\right), \qquad (3)$$

where sign is the well known three valued signum function. During recombination, for each component j of the integer mutant vector, u_{g+1}^i, a random real number, r, in the interval $[0, 1]$ is obtained and compared with the *crossover constant*, CR. If $r \leqslant CR$ we select as the j-th component of the trial vector, v_{g+1}^i, the corresponding component of the mutant vector, u_{g+1}^i. Otherwise, we choose the j-th component of the target vector, w_g^i. It must be noted that the result of this operation is also a 6-bit integer vector. Finally, the trial individual is accepted for the next generation only if it reduces the value of the objective function (selection operator).

Furthermore, the subpopulations are independently evolved in parallel and occasionally migration is employed to allow cooperation between them through the migration operator (see Section 4.3 below). The DDE algorithm is based of the asynchronous island model which is exhibited in Algorithm 1. Additionally, for completeness purposes let us briefly present the EA step of the island model in the case of DDE algorithm (Algorithm 2).

Algorithm 2 DE step in DDE algorithm

```
1: for Each individual $w_g^i$ in the subpopulation do
2:     Evaluate the fitness, $f(w_g^i)$
3:     Create a trial vector $u_{g+1}^i$ by applying a mutation operator
4:     Round $u_{g+1}^i$ to the nearest integer using Eq.(3)
5:     Create an offspring, $v_{g+1}^i$, by applying the crossover operator
6:     if $f(v_{g+1}^i) < f(w_g^i)$ then
7:         $w_{g+1}^i = v_{g+1}^i$
8:     else
9:         $w_{g+1}^i = w_g^i$
10:    end if
11: end for
```

4.2. The Distributed PSO algorithm

The Particle Swarm Optimization (PSO) algorithm is an Evolutionary Computation technique, which belongs to the category of Swarm Intelligence methods. It was introduced by Eberhart and Kennedy [40] in 1995. PSO is inspired by the social behavior of bird flocking and fish schooling, and is based on a social-psychological model of social influence and social learning. The fundamental hypothesis to the development of PSO is that an evolutionary advantage is gained through the social sharing of information among members of the same species. Furthermore, the behavior of the individuals of a flock corresponds to fundamental rules, such as nearest-neighbor velocity matching and acceleration by distance [8, 41]. Like DE, PSO is capable of handling non-differentiable, discontinuous and multimodal objective functions and has shown great promise in several real-world applications [15, 17, 18].

To this end, PSO is a population–based stochastic algorithm that exploits a population of individuals, to effectively probe promising regions of the search space. Thus, each individual (particle) of the population (swarm) moves with an adaptable velocity within the search space and retains in its memory the best position it ever encountered. There are two variants of PSO, namely the *global* and the *local*. In the *global* variant, the best position ever attained by all individuals of the swarm is communicated to all the particles, while in the *local* variant, for each particle it is assigned a neighborhood consisting of a pre-specified number of particles and the best position ever attained by the particles in their neighborhood is communicated among them [41].

More specifically, each particle is an D-dimensional vector, and the swarm consists of NP particles. Thus, the position the i-th particle of the swarm can be represented as: $X_i = (x_{i1}, x_{i2}, \ldots, x_{iD})$. The velocity of each particle is also an D-dimensional vector, and for the i-th particle is denoted as: $V_i = (u_{i1}, u_{i2}, \ldots, u_{iD})$. The best previous position of the i-th particle can be recorded as: $P_i = (p_{i1}, p_{i2}, \ldots, p_{iD})$, and the best particle in the swarm, the particle with the smallest fitness function value, is indicated by the index g. Furthermore, the neighborhood of each particle is usually defined through the particles' indices. The most common topology is the *ring* topology, where the neighborhood of each particle consists of particles with neighboring indices [42].

Clerc and Kennedy [43], proposed a version of PSO which incorporates a new parameter χ, known as the *constriction factor*. The main role of the constriction factor is to control the magnitude of the velocities and alleviate the "swarm explosion" effect that prevented the convergence of the original PSO algorithm [44]. According to [43], the dynamic behavior of the particles in the swarm is manipulated using the following equations:

$$\begin{aligned} V_i(t+1) &= \chi(V_i(t) + c_1 r_1 (P_i(t) - X_i(t)) + c_2 r_2 (P_g(t) - X_i(t))), && (4) \\ X_i(t+1) &= X_i(t) + V_i(t+1), && (5) \end{aligned}$$

where $i = 1, 2, \ldots, N$, c_1 and c_2 are positive constants referred to as *cognitive* and *social* parameters respectively, and r_1 and r_2 are randomly chosen numbers uniformly distributed in $[0, 1]$. The resulting position of the i-th particle ($X_i(t+1)$) is a real weight vector. To this end, similarly to the DDE implementation, we round $X_i(t+1)$ to the nearest integer and subsequently utilize equation (3) to constrain it in the range $[-32, 32]$.

In a stability analysis provided in [43] it was implied that the constriction factor is typically calculated according to the formula:

$$\chi = \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}, \tag{6}$$

for $\phi > 4$, where $\phi = c_1 + c_2$, and $k = 1$.

Furthermore, as Differential Evolution algorithm, the Particle Swarm Optimization algorithm can be easily parallelized. The incorporation of PSO into an island model is a straightforward procedure. Each island evolves in parallel a sub-swarm of particles and occasionally migration is employed to allow cooperation between them through the migration operator (Section 4.3). Notice that, here the integer weights w^i_{g+1} are represented with the $X_i(t+1)$ notation, where t denotes the number of the current generation and i represents the i-th particle of the sub-swarm. The DPSO algorithm is based of the asynchronous island model which is exhibited in Algorithm 1. Additionally, for completeness purposes let us briefly present the EA step of the island model in the case of DPSO algorithm (Algorithm 3).

Algorithm 3 PSO step in DPSO algorithm

```
1: for Each particle $X^i_t$ in the sub-swarm do
2:     Evaluate the fitness, $f(X^i_t)$
3:     Calculate the personal best position $P_i(t)$, and the neighborhood best
       position $P_g(t)$ based on the local or global variant of the algorithm
4: end for
5: for Each particle $X^i_t$ in the sub-swarm do
6:     Update the velocity of the particle $X^i_t$ using Eq.(4)
7:     Update the position of the particle $X^i_t$ using Eq.(5)
8: end for
```

Next, we briefly describe the operator controlling the migration of the best individuals.

4.3. The Migration Operator

The distributed versions of the DE and PSO algorithms have been employed according to the dEA paradigm. 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 migration of the best individuals is controlled by the migration constant φ. A good choice for the migration constant is one that allows each subpopulation to evolve for some iterations independently before the migration phase actually occur. There is a critical migration constant value below which the DDE and DPSO performance is hindered by the isolation of the subpopulations, and above which the subpopulations are able to locate solutions of the same quality as the panmictic implementations. Detailed description of the DDE algorithm and experimental results on difficult optimization problems can be found in [14, 22]. A parallel implementation of the PSO algorithm can be found in [45]. Next, we briefly describe the distributed PSO algorithm utilized in this study.

4.4. The Message Passing Interface

The Message Passing Interface (MPI) is a portable message-passing standard that facilitates the development of parallel applications and libraries. MPI is the specification resulting from the MPI-Forum [46] which involved several organizations designing a portable system which can allow programs to work on a heterogeneous network. MPI implementations for executing parallel applications run on both tightly-coupled massively-parallel machines and on networks of distributed workstations with separate memory. With this system, each executing process will communicate and share its data with others by sending and receiving messages. The MPI functions support process-to-process communication, group communication, setting up and managing communication groups, and interacting with the environment. Thus, MPI can be incorporated for dEA and/or cEA implementation.

A large number of MPI implementations are currently available, each of which emphasizes different aspects of high-performance computing or is intended to solve a specific research problem. In this paper the OpenMPI implementation of the MPI standard has been utilized. OpenMPI is open source, peer-reviewed, production-quality complete MPI implementation, which provides extremely high performance [47].

5. Experimental Results

In this study, the sequential, as well as the distributed versions of the DE and PSO algorithms are applied to train PSNs with integer weights and threshold activation functions. Here, we report results from the following well known and widely used neural network training problems:

1. N-bit Parity check problems [48, 49],
2. the numeric font classification problem (NumFont) [50],
3. the MONK's classification problems (MONK1, MONK2, and MONK3) [51],
4. the handwritten digits classification problem (PenDigits) [52], and

5. the rock vs. mine sonar problem (Sonar) [53].

For all the training problems, we have used the fixed values of $F = 0.5$ and $CR = 0.7$ as the DE mutation and crossover constants respectively. Similarly, for the PSO algorithm, fixed values for the cognitive and social parameters $c_1 = c_2 = 2.05$ have been used, and the constriction factor $\xi = 0.729$ has been calculated using Eq. (6).

Regarding the number of hidden neurons, we tried to minimize the degrees of freedom of the PSN. Thus, the simpler network topology, which is capable to solve each problem, has been chosen. Below we exhibit the experimental results from the sequential and the distributed DE and PSO implementations. For all the experiments reported below we utilize threshold activation functions and 6-bit integer weights.

5.1. Sequential DE and PSO Implementation

Here, we exhibit experimental results from the sequential DE and PSO algorithms. We call DE_1 and DE_2 the DE algorithms that use the mutation operators defined in equations (1) and (2), respectively. We call PSO_1 and PSO_2 the local and the global PSO variant, respectively. The neighborhood of each particle had a radius of one. Specifically, the neighborhood of the i-th particle contains the $(i-1)$-th and the $(i+1)$-th particles. Notice that the software used in this section does not contain calls to the MPI library. To this end, this implementation is marginally faster than the distributed implementation executed in only one computer node.

The first set of experiments consists of the N-bit parity check problems. These problems are well known and widely used benchmarks and are suitable for testing the non-linear mapping and "memorization" capabilities of neural networks. Although these problems are easily defined they are hard to solve, because of their sensitivity to initial weights and their multitude of local minima. Each N-bit problem has 2^N patterns with N attributes in each pattern. All patterns have been used for training and testing. For each N-bit problem we have used an N degree Pi–Sigma network (resulting N neurons in the middle layer). Here, we report results for $N = 2, 3, 4, 5$.

For each problem and each algorithm, we have used 10 individuals in each population and have conducted 1000 independent simulations. The termination criterion applied to the learning algorithm was the mean square training error (MSE) and it was different for each N-bit parity problem (0.05, 0.025, 0.125, and 0.125 respectively), following the experimental setup of [36]. Notice that the PSNs trained here have threshold activation functions.

Table 1 shows the experimental results for the parity check problems. The reported parameters for the simulations that have reached solution are: *Min* the minimum number, *Mean* the mean value, *Max* the maximum number, and *St.D.* the standard deviation of the number of training generations. All trained networks gave perfect generalization capabilities for all problems. The results of PSNs having threshold activation functions reported below are equivalent or better than the results of PSNs trained using the classical back-propagation algorithm [36]. An additional advantage of the proposed approach is that no gradient information is required; no backward passes were performed.

			Generations			
N	Topology	Algorithm	*Min*	*Mean*	*Max*	*St.D.*
2	2–2–1	DE_1	1	1.70	5	1.36
2	2–2–1	DE_2	1	5.04	12	4.92
2	2–2–1	PSO_1	1	1.92	10	1.26
2	2–2–1	PSO_2	1	2.07	13	1.71
3	3–3–1	DE_1	1	13.93	50	10.16
3	3–3–1	DE_2	1	17.98	77	13.95
3	3–3–1	PSO_1	1	23.21	177	21.91
3	3–3–1	PSO_2	1	29.06	281	35.28
4	4–4–1	DE_1	1	9.09	47	8.29
4	4–4–1	DE_2	1	9.66	34	8.55
4	4–4–1	PSO_1	1	2.02	10	1.42
4	4–4–1	PSO_2	1	2.20	17	1.74
5	5–5–1	DE_1	1	36.14	100	21.21
5	5–5–1	DE_2	1	35.98	100	21.76
5	5–5–1	PSO_1	1	27.01	200	28.51
5	5–5–1	PSO_2	1	28.53	210	29.74

Table 1: Simulation results for the N-bit parity check problem.

Below we report experimental results from the sequential DE and PSO implementations on (a) the numeric font, (b) the MONK's, (c) the handwritten digits and (d) the rock vs. mine sonar classification problems. To present the generalization results the following notation is used in the following Tables: *Min* indicates the minimum generalization capability of the trained PSNs; *Max* is the maximum generalization capability; *Mean* is the average generalization capability; *St.D.* is the standard deviation of the generalization capability. In all cases, average performance presented was validated using the well known test for statistical hypotheses, named t–test (see for example [54]), using the SPSS 15 statistical software package.

It must be noted that PSNs trained for the MONK1, MONK2, MONK3, and the Sonar training problems have only one output unit, since all the samples of those datasets belong to one of the two available classes. On the other hand, the networks trained for the NumFont and the PenDigits classification problems have ten output units (one for each digit). To implement a PSN having multiple output units is equivalent to constructing PSNs having common input units and different middle layer units (thus, different sets of weights), each having one output unit. Thus, a PSN should be trained to discriminate samples from each problem class.

5.1.1. The Numeric Font Classification Problem

For the numeric font classification problem the aim is to train a PSN to recognize 8×8 pixel machine printed numerals from zero to nine in standard helvetica font [50]. After being trained, the PSN was tested for its generalization capability using helvetica italic font. Note that, the test patterns in the italic font have 6 to 14 bits reversed from the training patterns. To evaluate the average generalization performance the *max* rule was used.

For the NumFont problem we trained 10 distinct PSNs, each one having 16 input units and one output unit. Thus, one PSN for each digit has been trained and we have conducted 1000 independent simulations for each network. The termination criterion applied to the learning algorithm was either a training error less than 0.001 or 1000 iterations. The experimental results are presented in Table 2. All algorithms exhibited good generalization capabilities. DE_1 in particular achieved 100% generalization success, followed closely by PSO_2. This indicates that the global variants exhibited better results for this problem.

Network	Mutation	Generalization (%)			
Topology	Strategy	*Min*	*Mean*	*Max*	*St.D.*
64–1–1	DE_1	80	99.4	100	2.50
64–1–1	DE_2	100	100	100	0.00
64–1–1	PSO_1	80	95.9	100	5.70
64–1–1	PSO_2	90	99.8	100	1.21

Table 2: Generalization results for the NumFont problem.

5.1.2. The MONK's Classification Problems

The MONK's classification problems are three binary classification tasks, which have been used for comparing the generalization performance of learning algorithms [51]. These problems 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 [51], for details). Each possible value for every attribute is assigned a single bipolar input, resulting 17 inputs.

For the MONK's problems we have tested PSNs having two units in the middle layer (i.e. 17–2–1 PSN architecture). Table 3 illustrates the average generalization results (1000 runs were performed). The termination criterion applied to the learning algorithm was either a training error less than 0.01 or 5000 iterations. Once again the DE and PSO trained PSNs exhibited high classification success rates, while the training procedure was very fast and robust. Notice that it has been theoretically proved that PSNs are capable to learn perfectly any Boolean Conjunctive Normal Form (CNF) expression [37] and that the MONK's problems can be described in CNF.

			Generalization (%)			
Problem	Topology	Algorithm	*Min*	*Mean*	*Max*	*St.D.*
MONK1	17–2–1	DE_2	86	96.68	100	2.43
MONK1	17–2–1	DE_2	86	96.74	100	2.38
MONK1	17–2–1	PSO_1	80	95.16	100	3.30
MONK1	17–2–1	PSO_2	83	96.02	100	2.66
MONK2	17–2–1	DE_2	79	97.36	100	2.38
MONK2	17–2–1	DE_2	91	97.66	100	1.45
MONK2	17–2–1	PSO_1	90	96.86	100	1.69
MONK2	17–2–1	PSO_2	91	97.31	100	1.64
MONK3	17–2–1	DE_2	82	91.57	97	2.37
MONK3	17–2–1	DE_2	81	90.77	97	3.10
MONK3	17–2–1	PSO_1	80	92.02	99	2.97
MONK3	17–2–1	PSO_2	81	93.14	99	2.46

Table 3: Generalization results for the MONK's Problems

5.1.3. The Handwritten Digits Classification Problem

The PenDigits problem is part of the UCI Repository of Machine Learning Databases [52] and is characterized by a real–valued training set of approximately 7500 patterns. In this experiment, a digit database has been assembled by collecting 250 samples from 44 independent writers. The samples written by 30 writers are used for training, and the rest are used for testing. The training set consists of 7494 real valued samples and the test set of 3498 samples.

For the PenDigits problem we trained 10 different PSNs, one PSN for each digit. We have conducted 100 independent simulations for each network and the termination criterion applied to the learning algorithm was either a training error less than 0.001 or 1000 iterations. Table 4 exhibits the average generalization results. The average classification accuracy of the trained PSNs for the PenDigits problem is about 85% for all algorithms. The average classification accuracy of the trained PSNs for the PenDigits

Network	Mutation	Generalization (%)			
Topology	Strategy	*Min*	*Mean*	*Max*	*St.D.*
16–2–1	DE_1	83.91	86.20	88.74	1.08
16–2–1	DE_2	81.53	84.60	87.71	1.16
16–2–1	PSO_1	82.38	84.76	87.19	1.20
16–2–1	PSO_2	82.59	85.16	87.70	1.17

Table 4: Generalization results for the PenDigits Problem.

problem is about 85% for all algorithms.

5.1.4. The Sonar Problem

For the Sonar problem the task is to train a PSN to discriminate between sonar signals bounced off a metal cylinder (mine) and those bounced off a roughly cylindrical rock. In this experiment the dataset contains 208 samples obtained by bouncing sonar signals off a metal cylinder and a rock at various angles and under various conditions [53]. There exist 111 samples obtained from mines and 97 samples obtained from rocks. Each pattern consists of 60 real numbers in the range $[0.0, 1.0]$. Each number represents the energy within a particular frequency band, integrated over a certain period of time. The trained PSNs have one unit in the middle layer (i.e. 60–1–1 PSN architecture).

The classification accuracy of the trained PSNs is exhibited in Table 5. The average classification accuracy obtained by the EA trained PSNs is comparable to the classification accuracy of FNNs.

Network	Mutation	Generalization (%)			
Topology	Strategy	*Min*	*Mean*	*Max*	*St.D.*
60–1–1	DE_1	58	73.81	87	4.24
60–1–1	DE_2	57	73.35	87	4.34
60–1–1	PSO_2	64	73.89	85	3.92
60–1–1	PSO_1	61	74.44	90	3.85

Table 5: Generalization results for the Sonar problem.

5.2. Distributed DE and PSO Implementations

In this section the DDE and the DPSO algorithms are applied to train PSNs with integer weights and threshold activation functions. Here, we report results on the MONK's [51] as well as on the Sonar [53] benchmark problems.

For this set of experiments, we have conducted 1000 independent simulations for each algorithm, using a distributed computation environment consisting of $1, 2, 4, 6$, and 8 nodes. For the DDE and DPSO algorithms, we have used the same values for the algorithms' parameters. The migration constant was $\varphi = 0.1$. The termination criterion applied to the learning algorithm was either a training error less than 0.01 or 5000 iterations.

As for the choice of the communication topology, islands with many neighbors are more effective than sparsely connected ones. However, this brings forth a tradeoff between computation and communication cost. Optimal choice of the degree of the topology that minimizes the total cost is difficult. For the DDE and the DPSO implementation we have used the ring topology (each node communicates only with the next node on a ring).

In the distributed implementation, each processor evolves a subpopulation of potential solutions. To allow cooperation between the subpopulations, migration is employed. When a higher number of CPUs is utilized (i.e. higher number of subpopulations) the average generalization accuracy is slightly improved. This is probably due to the island model for the migration of the best individuals [14, 22].

The experimental generalization results of problems MONK1 and MONK2 are exhibited in Table 6, while the results of problems MONK3 and SONAR are presented in Table 7. Overall, the results indicate that the training of PSNs with integer weights and thresholds, using the modified DDE and DPSO algorithms are efficient and promising. The learning process was robust, fast and reliable, and the performance of the distributed algorithms

Computer	Mutation	MONK1 Generalization (%)				MONK2 Generalization (%)			
Nodes	Strategyi	*Min*	*Mean*	*Max*	*St.D.*	*Min*	*Mean*	*Max*	*St.D.*
1	DDE_1	90	97.26	100	2.18	93	98.00	100	1.42
1	DDE_2	89	97.44	100	2.06	94	98.12	100	1.39
1	$DPSO_1$	85	96.16	100	2.54	91	97.14	100	1.62
1	$DPSO_2$	91	97.59	100	1.71	93	98.19	100	1.23
2	DDE_1	90	97.81	100	2.18	94	97.88	100	1.36
2	DDE_2	89	97.84	100	1.73	93	97.74	100	1.56
2	$DPSO_1$	88	96.75	100	2.47	93	97.59	100	1.50
2	$DPSO_2$	90	97.59	100	1.92	96	98.35	100	1.19
4	DDE_1	93	98.13	100	1.79	93	98.12	100	1.14
4	DDE_2	91	97.90	100	1.93	93	98.00	100	1.46
4	$DPSO_1$	93	97.27	100	1.88	93	97.90	100	1.46
4	$DPSO_2$	93	97.87	100	1.49	95	98.21	100	1.12
6	DDE_1	91	97.86	100	1.91	94	98.12	100	1.25
6	DDE_2	92	97.73	100	1.85	94	97.79	100	1.28
6	$DPSO_1$	92	96.78	100	1.69	93	97.61	100	1.49
6	$DPSO_2$	92	97.80	100	1.68	95	98.18	100	1.22
8	DDE_1	92	98.22	100	1.59	95	97.96	100	1.33
8	DDE_2	93	97.77	100	1.59	95	98.24	100	1.15
8	$DPSO_1$	90	97.05	100	2.03	92	97.48	100	1.71
8	$DPSO_2$	94	97.91	100	1.26	95	98.19	100	1.08

Table 6: Generalization results for the MONK1 and MONK2 benchmark problems

stable. Additionally, the trained PSNs utilizing DDE and DPSO exhibited good generalization capabilities.

The four methods considered here, exhibit similar performance. To better compare them, we have performed ANOVA tests and post hoc analysis (Tukey). For the 8 computer node case, the statistical results indicate that in the MONK problems the four methods exhibit different behavior, while they are equivalent in the SONAR problem. More specifically, in the MONK1 problem the two PSO variants are equivalent, while in the MONK2 the global methods (i.e. DDE_2 and $DPSO_2$) are equivalent.

In addition to the generalization accuracy test, we have also compared the four methods by means of the time needed to train the PSNs.

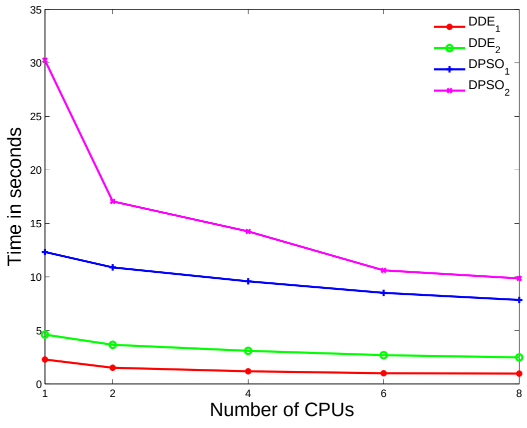
Computer	Mutation	MONK3 Generalization (%)				SONAR Generalization (%)			
Nodes	Strategy	*Min*	*Mean*	*Max*	*St.D.*	*Min*	*Mean*	*Max*	*St.D.*
1	DDE_1	83	92.69	97	2.19	61	76.62	88	5.87
1	DDE_2	81	91.06	97	2.98	60	76.49	90	6.08
1	$DPSO_1$	84	92.16	97	2.79	62	76.37	91	5.81
1	$DPSO_2$	86	92.90	98	2.42	60	77.16	90	5.46
2	DDE_1	87	92.49	97	2.14	54	76.22	90	5.87
2	DDE_2	82	90.99	96	2.62	62	76.40	90	5.63
2	$DPSO_1$	82	91.55	96	2.61	64	76.97	88	5.25
2	$DPSO_2$	83	91.99	98	3.04	62	77.55	90	5.82
4	DDE_1	83	92.37	97	2.47	61	75.90	90	6.23
4	DDE_2	82	90.40	97	3.14	64	76.05	88	5.32
4	$DPSO_1$	83	90.91	97	3.00	58	76.77	91	6.42
4	$DPSO_2$	85	92.80	98	2.48	62	76.90	88	5.54
6	DDE_1	84	92.71	96	2.11	58	76.59	87	5.92
6	DDE_2	83	90.45	96	3.22	58	75.89	87	6.20
6	$DPSO_1$	84	91.27	96	2.85	58	76.47	90	6.04
6	$DPSO_2$	82	91.67	98	3.33	61	75.48	90	5.66
8	DDE_1	85	92.34	96	1.93	61	76.53	87	5.60
8	DDE_2	82	90.64	95	2.66	60	76.06	91	5.83
8	$DPSO_1$	84	90.94	96	2.79	61	77.00	90	5.83
8	$DPSO_2$	80	92.51	97	2.67	60	77.16	90	6.26

Table 7: Generalization results for the MONK3 and SONAR benchmark problems

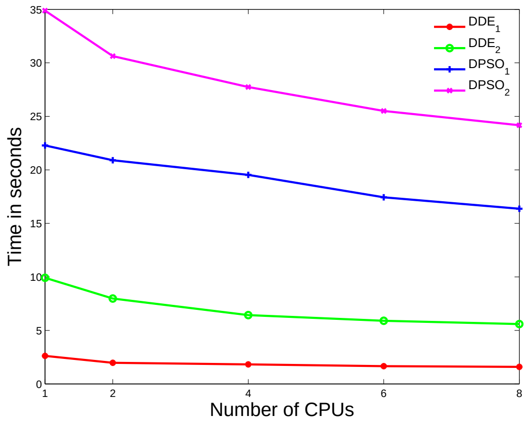
5.2.1. Distributed DE and PSO Times and Speedup Measurements

To better understand the efficiency of the proposed methods we have measured the time needed to converge to a solution. Figure 2 illustrates average elapsed wall-clock times. For every experiment, the MPI timer (MPI Wtime) was used. This procedure is a high-resolution timer, which calculates the elapsed wall-clock time between two successive function calls. From the results, it is evident that the DDE algorithms are faster and trained the PSNs efficiently. Among the DDE algorithms DDE_1 is marginally better, while $DPSO_1$ and $DPSO_2$ seem equivalent.

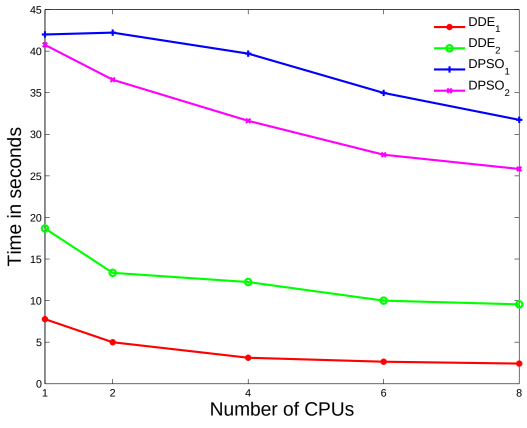
Notice that DDE_1 mutation operator uses the best individual of the current generation for computation the mutant vector. On the other hand, DDE_2 computes the mutant vector from randomly chosen individuals. To this end, DDE_1 converges faster to a single minimum, while DDE_2 better explores the search space. Similarly, $DPSO_1$ is the local version of PSO, while $DPSO_2$ is the global version. Thus, to train a HONN for a new application, where the balance between exploration and exploitation is unknown, both local and global algorithms can be tried. Furthermore, one can start the training process using the DDE_2 or the $DPSO_2$ for better exploration and consequently switch to DDE_1 or $DPSO_1$ for faster convergence [23]. If only one algorithm must be utilized, in the case of an unknown problem, we recommend the use of the DDE_1 algorithm.



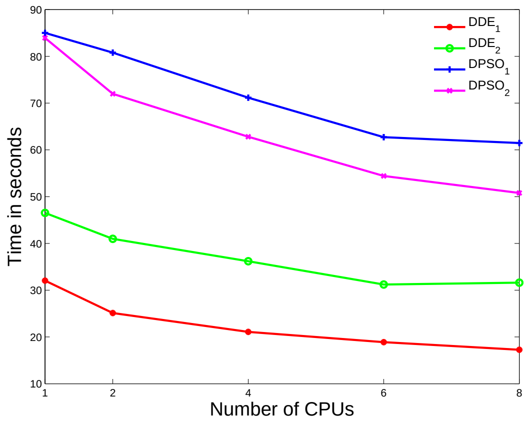
a. MONK1 problem



b. MONK2 problem



c. MONK3 problem



d. SONAR problem

Figure 2: Average elapsed wall-clock times for training PSNs by DDE_1, DDE_2, $DPSO_1$ and $DPSO_2$, for the MONK's and the Sonar problems

In addition to time measurements, we also calculated the speedup achieved by assigning each subpopulation to a different processor relative to assigning all subpopulations to a single processor. The speedup is illustrated in Figure 3. In the literature various speedup measurement methods have been proposed. However, to perform fair comparison between the sequential and the parallel (or distributed) code, several conditions must be met [32, 55]:

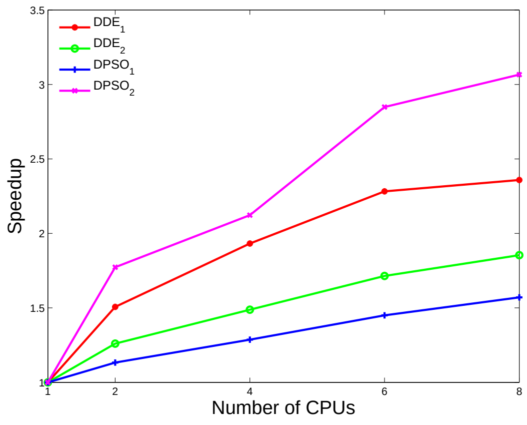
1. average and not absolute times must be used,
2. the uni- and multi-processor implementations should be exactly the same, and
3. the parallel (or distributed) code must be run until a solution for the problem is found.

To obtain the plotted values, we conducted 1000 independent simulations for 1, 2, 4, 6, 8 computer nodes and the average speedup is shown. For every simulation the training error goal was met and the migration constant was equal to 0.1.

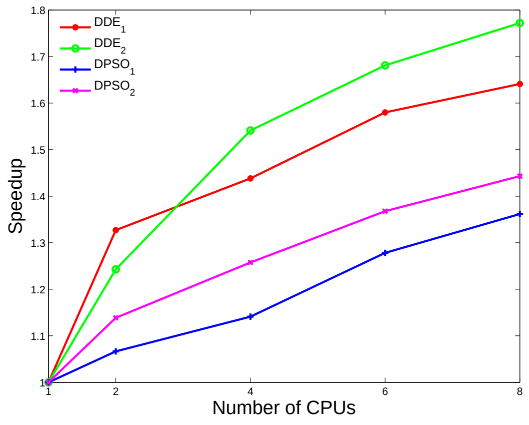
Several factors can influence the speedup, such as the local area network load and the CPU load due to system or other users' tasks [32, 56, 57]. Nevertheless, the speedup results indicate that the combined processing power overbalances the overhead due to process communication and speedup is achievable. It must be noted that the DDE_1 and DPSO_2 generally exhibit higher speedup results, with DDE_1 being the best parallelized algorithm. Overall, the best speedup was achieved by DDE_1 on the MONK3 problem, when 8 computer nodes were utilized (approximately 3.2 times faster than the simulation utilizing one computer node). Once again the use of DDE_1 is recommended for large distributed systems.

6. Concluding Remarks

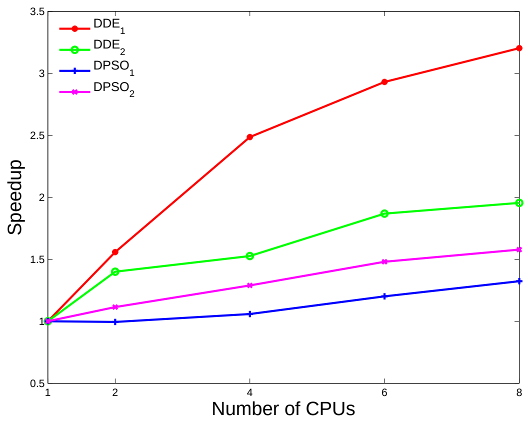
In this paper, we study a special class of Higher-Order Neural Networks, the Pi–Sigma Networks and propose the use of sequential as well as parallel (or distributed) Evolutionary Algorithms for their training. The incorporation of global optimization methods (such as Evolutionary Algorithms) instead of classical local optimization methods is strongly recommended. Global optimization methods incorporate efficient and effective searching mechanisms that avoid the convergence to local minima and thus enhance the neural network training procedure, as well as the classification accuracy of the trained networks. Additionally, EAs' capabilities of handling discrete, non-differentiable, discontinuous and multimodal objective functions, provide the ability to apply them for training "hardware–friendly" PSNs, i.e. PSNs with threshold activation functions and small integer weights.



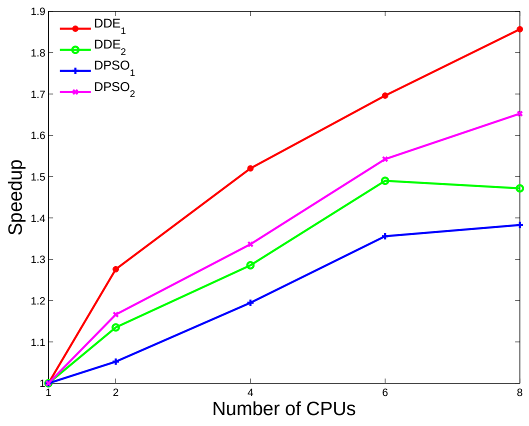
a. MONK1 problem



b. MONK2 problem



c. MONK3 problem



d. SONAR problem

Figure 3: Speedup of training PSNs by DDE$_1$, DDE$_2$, DPSO$_1$ and DPSO$_2$, for the MONK's and the Sonar problems

For the proposed distributed versions of Differential Evolution and Particle Swarm Optimization algorithms each processor of a distributed computing environment is assigned a subpopulation of potential solutions. The subpopulations are independently evolved in parallel and occasional migration of the best individuals is employed to allow subpopulation cooperation. Such parallel or distributed EAs implementations enhances the training process of the Pi–Sigma Networks, due to the parallel search of the solution space, while they speedup the training process due to the usage of multiple CPUs.

The performance of the trained networks is evaluated through well known neural network training problems and the experimental results suggest that the proposed training approach using distributed Evolutionary Algorithms is robust, reliable, and efficient. By assigning each subpopulation to a different processor significant training speedup was achieved (approximately 3.2 times faster than the sequential implementation). The trained networks were able to effectively address several difficult classification tasks. Moreover, the EA trained PSNs exhibited good generalization capabilities, comparable with the best generalization capability of PSNs trained using other well–known batch training algorithms, such as the BP and the RProp [58]. Among the EA algorithms studied, the local variant of the DE algorithm (DDE_1) was clearly the fastest one. Thus, the use of DDE_1, in an unknown optimization task, is recommended.

Finally, it has to be noted that the incorporation of either small integer weights or threshold activation functions did not hindered the performance and the generalization capabilities of the Pi-Sigma Networks. Furthermore, in a future communication we intend to rigorously compare the classification capability of PSNs with other soft computing approaches, as well to tackle real-world problems with smaller integer range weights. Additionally, we will give experimental results of PSNs trained using hierarchical parallel Evolutionary Algorithms.

Acknowledgements

The authors would like to thank the editor and the anonymous reviewers for their useful comments and suggestions that helped to improve the content as well as the clarity of the manuscript.

This work was partially supported by an “Empirikion Foundation” award that helped the acquisition and the setup of the distributed computer cluster.

References

[1] D. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison Wesley, Reading, MA, 1989.

[2] J. H. Holland, Adaptation in natural and artificial system, University of Michigan Press, 1975.

[3] D. Fogel, Evolutionary Computation: Towards a New Philosophy of Machine Intelligence, IEEE Press, Piscataway, NJ, 1996.

[4] L. J. Fogel, A. J. Owens, M. J. Walsh, Artificial intelligence through simulated evolution, Wiley, 1966.

[5] N. Hansen, A. Ostermeier, Completely derandomized self-adaptation in evolution strategies, Evolutionary Computation 9 (2) (2001) 159–195.

[6] I. Rechenberg, Evolution strategy, in: J. Zurada, R. Marks II, C. Robinson (Eds.), Computational Intelligence: Imitating Life, IEEE Press, Piscataway, NJ, 1994.

[7] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, USA, 1992.

[8] J. Kennedy, R. C. Eberhart, Particle swarm optimization, in: Proceedings IEEE International Conference on Neural Networks, Vol. IV, IEEE Service Center, Piscataway, NJ, 1995, pp. 1942–1948.

[9] R. Storn, K. Price, Differential evolution – a simple and efficient adaptive scheme for global optimization over continuous spaces, Journal of Global Optimization 11 (1997) 341–359.

[10] T. Baeck, D. B. Fogel, Z. Michalewicz (Eds.), Handbook of Evolutionary Computation, Oxford University Press, 1997.

[11] Y. Shin, J. Ghosh, The pi–sigma network: An efficient higher-order neural network for pattern classification and function approximation, in: International Joint Conference on Neural Networks, 1991.

[12] D. S. Huang, H. H. S. Ip, K. C. K. Law, Z. Chi, Zeroing polynomials using modified constrained neural network approach, IEEE Transactions on Neural Networks 16 (3) (2005) 721–732.

[13] S. Perantonis, N. Ampazis, S. Varoufakis, G. Antoniou, Constrained learning in neural networks: Application to stable factorization of 2-d polynomials, Neural Process. Lett. 7 (1) (1998) 5–14.

[14] V. P. Plagianakos, M. N. Vrahatis, Parallel evolutionary training algorithms for 'hardware–friendly' neural networks, Natural Computing 1 (2002) 307–322.

[15] M. Clerc, Particle Swarm Optimization, ISTE Publishing Company, 2006.

[16] G. Magoulas, V. Plagianakos, M. Vrahatis, Neural network-based colonoscopic diagnosis using on-line learning and differential evolution, Applied Soft Computing 4 (2004) 369–379.

[17] K. Parsopoulos, M. Vrahatis, On the computation of all global minimizers through particle swarm optimization, IEEE Transactions on Evolutionary Computation 8 (3) (2004) 211–224.

[18] K. E. Parsopoulos, M. N. Vrahatis, Recent approaches to global optimization problems through particle swarm optimization, Natural Computing: an international journal 1 (2-3) (2002) 235–306.

[19] V. Plagianakos, G. Magoulas, M. Vrahatis, Evolutionary training of hardware realizable multilayer perceptrons, Neural Computing and Application 15 (2005) 33–40.

[20] V. P. Plagianakos, M. N. Vrahatis, Neural network training with constrained integer weights, in: P. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, A. Zalzala (Eds.), Congress of Evolutionary Computation (CEC'99), IEEE Press, Washington D.C., U.S.A., 1999, pp. 2007–2013.

[21] R. Storn, System design by constraint adaptation and differential evolution, IEEE Transactions on Evolutionary Computation 3 (1999) 22–34.

[22] D. K. Tasoulis, N. G. Pavlidis, V. P. Plagianakos, M. N. Vrahatis, Parallel differential evolution, in: IEEE Congress on Evolutionary Computation (CEC 2004), 2004.

[23] D. K. Tasoulis, V. P. Plagianakos, M. N. Vrahatis, Clustering in evolutionary algorithms to efficiently compute simultaneously local and global minima, in: IEEE Congress on Evolutionary Computation (CEC 2005), Vol. 2, 2005, pp. 1847–1854.

[24] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (5) (2002) 443–462.

[25] E. Cantú-Paz, A survey of parallel genetic algorithms, Calculateurs Parallèles, Réseaux et Systèmes Répartis 10 (2) (1998) 141–171.

[26] E. Cantú-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[27] M. Gorges-Schleuter, Explicit parallelism of genetic algorithms through population structures, in: PPSN I: Proceedings of the 1st Workshop on Parallel Problem Solving from Nature, Springer-Verlag, London, UK, 1991, pp. 150–159.

[28] S. Gustafson, E. K. Burke, The speciating island model: an alternative parallel evolutionary algorithm, J. Parallel Distrib. Comput. 66 (8) (2006) 1025–1036.

[29] J. Sprave, A unified model of non-panmictic population structures in evolutionary algorithms, in: P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, A. Zalzala (Eds.), Proceedings of the Congress on Evolutionary Computation, Vol. 2, IEEE Press, 1999, pp. 1384–1391.

[30] Z. Skolicki, An analysis of island models in evolutionary computation, in: GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation, ACM, New York, NY, USA, 2005, pp. 386–389.

[31] Z. Skolicki, K. D. Jong, The influence of migration sizes and intervals on island models, in: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, New York, NY, USA, 2005, pp. 1295–1302.

[32] E. Alba, Parallel evolutionary algorithms can achieve super-linear performance, Information Processing Letters 82 (1) (2002) 7–13.

[33] R. Tanese, Parallel genetic algorithms for a hypercube, in: Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application, Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1987, pp. 177–183.

[34] E. Alba, J. M. Troya, A survey of parallel distributed genetic algorithms, Complex. 4 (4) (1999) 31–52.

[35] A. Y. H. Zomaya (Ed.), Parallel and Distributed Computing Handbook, McGraw-Hill, Inc., New York, NY, USA, 1996.

[36] J. Ghosh, Y. Shin, Efficient higher-order neural networks for classification and function approximation, in: Int. J. Neural Systems, Vol. 3, 1992, pp. 323–350.

[37] Y. Shin, J. Ghosh, Realization of boolean functions using binary pi-sigma networks, in: C. H. Dagli, S. R. T. Kumara, Y. C. Shin (Eds.), Intelligent Engineering Systems through Artificial Neural Networks, ASME Press, 1991, pp. 205–210.

[38] A. J. Hussain, P. Liatsis, Recurrent pi-sigma networks for dpcm image coding, Neurocomputing 55 (1-2) (2003) 363 – 382, support Vector Machines.

[39] K. Price, R. M. Storn, J. A. Lampinen, Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[40] R. C. Eberhart, J. Kennedy, A new optimizer using particle swarm theory, in: In Proceedings 6th Symposium on Micro Machine and Human Science, Nagoya, Japan, 1995, pp. 39–43.

[41] R. Eberhart, P. Simpson, R. Dobbins, Computational intelligence PC tools, Academic Press Professional, Inc., San Diego, CA, USA, 1996.

[42] R. Mendes, J. Kennedy, J. Neves, The fully informed particle swarm: simpler, maybe better, Evolutionary Computation, IEEE Transactions on 8 (3) (2004) 204–210.

[43] M. Clerc, J. Kennedy, The particle swarm - explosion, stability, and convergence in a multidimensional complex space, Evolutionary Computation, IEEE Transactions on 6 (1) (2002) 58–73.

[44] P. J. Angeline, Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences, in: EP '98: Proceedings of the 7th International Conference on Evolutionary Programming VII, Springer-Verlag, London, UK, 1998, pp. 601–610.

[45] N. Nedjah, E. Alba, L. de Macedo Mourelle, Parallel Evolutionary Computations (Studies in Computational Intelligence), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[46] M. P. I. Forum, Mpi: A message-passing interface standard, Tech. Rep. UT-CS-94-230 (1994).

[47] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.

[48] M. E. Hohil, D. Liu, S. H. Smith, Solving the n-bit parity problem using neural networks, Neural Networks 12 (9) (1999) 1321–1323.

[49] D. E. Rumelhart, J. L. McClelland, the PDP Research Group (Eds.), Parallel Distributed Processing - Vol. 1, The MIT Press, 1987.

[50] G. D. Magoulas, M. N. Vrahatis, G. S. Androulakis, Effective backpropagation training with variable stepsize, Neural Networks 10 (1) (1997) 69–82.

[51] S. B. Thrun, et.al., The MONK's problems: A performance comparison of different learning algorithms, Tech. Rep. CS-91-197, Pittsburgh, PA (1991).

[52] P. M. Murphy, D. W. Aha, Uci repository of machine learning databases (1994).

[53] R. P. Gorman, T. J. Sejnowski, Analysis of hidden units in a layered network trained to classify sonar targets, Neural Networks 1 (1988) 75–89.

[54] A. M. Law, W. D. Kelton, Simulation Modeling and Analysis, third edition Edition, McGraw-Hill, New York, 2000.

[55] W. Punch, How effective are multiple populations in genetic programming, in: J. e. a. Koza (Ed.), Proceedings of the Third Annual Conference on Genetic Programming, Morgan Kaufmann, Madison, WI, USA, 1998, pp. 308–313.

[56] J. He, X. Yao, Analysis of scalable parallel evolutionary algorithms, in: CEC 2006: IEEE Congress on Evolutionary Computation, 2006, pp. 120–127.

[57] J. I. Hidalgo, J. Lanchares, F. F. de Vega, D. Lombraina, Is the island model fault tolerant?, in: GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation, ACM, New York, NY, USA, 2007, pp. 2737–2744.

[58] M. G. Epitropakis, V. P. Plagianakos, M. N. Vrahatis, Higher–order neural networks training using differential evolution, in: International Conference of Numerical Analysis and Applied Mathematics, Wiley-VCH, Crete, Greece, 2006, pp. 376–379.