Professor Vasile GEORGESCU, PhD
University of Craiova
E-mail: vasile.georgescu@feaa.ucv.ro

USING NATURE-INSPIRED METAHEURISTICS TO TRAIN PREDICTIVE MACHINES

Abstract. *Nature-inspired metaheuristics for optimization have proven successful, due to their fine balance between exploration and exploitation of a search space. This balance can be further refined by hybridization. In this paper, we conduct experiments with some of the most promising nature-inspired metaheuristics, for assessing their performance when using them to replace backpropagation as a learning method for neural networks. The selected metaheuristics are: Cuckoo Search (CS), Gravitational Search Algorithm (GSA), Particle Swarm Optimization (PSO), the PSO-GSA hybridization, Many Optimizing Liaisons (MOL) and certain combinations of metaheuristics with local search methods. Both the neural network based classifiers and function approximators are evolved in this way. Classifiers have been evolved against a training dataset having bankruptcy prediction as a target, whereas function approximators have been evolved as NNARX models, where the target is to predict foreign exchange rates.*

Keywords: *Nature inspired metaheuristics; Hybridizations; Training Neural Networks with metaheuristics, instead of backpropagation; Classifiers; Function Approximators; Bankruptcy prediction*; *Prediction with NNARX models.*

JEL classification: C22, C45, C51, C53, C63 G17

1. NATURE-INSPIRED METAHEURISTICS FOR OPTIMIZATIONS

Heuristics are strategies using readily accessible, though loosely applicable, information to solve particular problems ([7]). In contrast to heuristics, meta-heuristics designate some form of stochastic computational approach to an optimization problem, and consist of iteratively searching for a solution that is "good enough" (with regard to a given measure of quality), over a very large set of candidate solutions. Although metaheuristics are general-purpose methods, they still need some fine-tuning of their behavioral parameters in order to adapt the technique to the problem at hand.

The trade-off between the collection of new information (exploration) and the use of existing information (exploitation) is the key issue of any metaheuristic. It ensures the identification of new promising regions in the search space to escape

being trapped in local solutions, as well as the use of promising regions locally, to search for eventually reaching the global optimum. However, a common drawback of any metaheuristic is to not be as fast as local-search techniques, when it comes to exploitation, suffering from low convergence rate in the last stage of approaching the solution. It is these complementary strengths that inspired the use of hybridization for achieving a good balance between exploration and exploitation. Hybridization has been first advocated in a paper of Eiben and Schippers ([1], 1998) and, since then, it became more and more influential, leading to the development of numerous hybrid metaheuristics, in hoping that the hybrids perform better than the individual algorithms. Exploration is sometimes associated with diversification, whereas exploitation is associated with intensification. Usually, an equilibrium is insured by favoring exploration at the beginning of the search (when it is desirable to have a high level of diversification) and favoring exploitation at the end (when the algorithm is close to the final solution and intensifying the local search is more suitable).

Certain predictive machines have been proven to be universal approximators; among them, multilayer perceptrons (Hornick et al., 1989) and fuzzy systems (Kosko, 1992) are the most notorious.

It is worth noticing that feedforward neural networks achieved their status of universal approximators due to the introduction of Back-Propagation (BP) as a training method. The standard version of the algorithm looks for the minimum of the error function in the weight space using the gradient descent method. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. However, the success and speed of training depends upon the initial parameter settings, such as architecture, initial weights and biases, learning rates, and others. Actually, this need for an ex-ante specification of the NN architecture and various initial parameters is one of the main drawbacks of BP. Imposing the differentiability condition on transfer functions is another one.

Such drawbacks have motivated an increasing interest in alternative methods, able to automatically evolve feedforward neural networks under less restrictive conditions than training the network with back-propagation.

By their very nature as global optimization tools, metaheuristics can be seen as good replacements for back-propagation, because of the large size, nondifferentiability, complexity, and multimodality of the search space involved in training the network. The idea of evolving NN by evolutionary algorithms, as an alternative to BP, can be traced back to the late 1980s, when the emphasis was put on Genetic Algorithms (GAs). They have been used for evolving the connection weights with fixed network architecture, or for selecting the right network architecture. Occasionally, they have been used for more than one purpose – for example, evolving the network weights and the topology (structure) simultaneously. Different approaches have been used to encode the weights into the chromosome of a GA, including direct encoding schemes, in which each weight is explicitly represented in the chromosome, and indirect schemes, in which a compression scheme is used that requires an expansion of the chromosome to

derive the individual weights. Using GAs versus BP proved to have advantages and disadvantages. BP takes more time to reach the neighborhood of an optimal solution, but then reaches it more precisely. On the other hand, GAs investigate the entire search space. Hence, they reach faster the region of optimal solutions, but have difficulties to localize the exact point.

Meanwhile, a large number of nature-inspired metaheuristics have been proposed, and there has been an increasing interest in investigating the synergic effects of their hybridization. As general-purpose, global optimization algorithms, metaheuristics give rise to new opportunities, since they can address a variety of tasks and goals that cannot be achieved by BP. One such goal is to evolve, simultaneously, all the characteristics of a neural network. For example, in addition to the network weights and/or topological characteristics, one can also evolve the parameters of transfer functions. More precisely, given a sigmoidal transfer function, $y = 1/(1 + e^{-k \cdot input})$, the parameter k can be evolved along with the other characteristics of interest. One can also consider neural networks with nondifferentiable (even discontinuous) transfer functions, or with different transfer functions, for different neurons in the same layer. As for the fitness function of the evolved neural architecture, it can be specifically defined, in a way appropriate for the problem. It can thus incorporate variables helping to adjust the speed of learning, or the topological complexity of the network. Furthermore, there is the possibility of using a secondary metaheuristic (often called meta-evolution, or hyper-heuristic) as a meta-optimization procedure, in view of finding good performing behavior parameters for the primary metaheuristic. This parameter tuning stage is attempted to improve the ability of the primary metaheuristic to approach the global optimum when training the network.

NNs are well suited for both the classification and function approximation. This paper aims at assessing the performance of some of the most promising nature-inspired metaheuristics when using them to evolve classifiers as well as function approximators. The metaheuristics involved in our experiments are: Cuckoo Search (CS), Gravitational Search Algorithm (GSA), Particle Swarm Optimization (PSO), the PSO-GSA hybridization, Many Optimizing Liaisons (MOL) and certain combinations of metaheuristics with local search methods. The evolved NNs are then applied to predicting bankruptcy and foreign exchange rates.

2. METAHEURISTICS AND HYBRIDIZATIONS OF INTEREST

2.1. The Cuckoo Search Algorithm

Cuckoo search (CS) is a nature-inspired metaheuristic algorithm for optimization, developed by Yang and Deb ([13], 2009). It was inspired by cuckoos' breeding behavior, which typically consists of brood parasitism and nest takeover and may include the eviction of host eggs by recently hatched cuckoo chicks. CS was enhanced by the so-called Lévy flight behavior associated with some birds. In the meantime, a balanced combination of a local random walk with

permutation and the global explorative random walk is used, as a refined survival mechanism.

CS is a population-based algorithm, in a way similar to GAs and PSO, but it uses some sort of elitism and/or selection. Like other population-based algorithms, CS use reproduction operators to explore the search space. Each individual (i.e., egg) represents a solution to the problem under consideration. If the cuckoo egg mimics very well the host's, then it has the chance to survive and be part of the next generation. Exploring new and potentially better solutions is the main objective of the algorithm. The randomization in CS is more efficient as the step length is heavy-tailed, and any large step is possible. Another important characteristic of this heuristic is that it dependents only on a relatively small number of parameters. Actually, the number of parameters in CS to be tuned is fewer than in GA and PSO.

Recent experiments suggest that CS has the potential of outperforming PSO and GA in terms of predictive power. Moreover, given that each nest can represent a set of solutions, CS can be also extended to the type of meta-population algorithms.

Since animals search for food in a random or quasi-random manner, their foraging path is effectively a random walk: the next move is based on the current location or state and the transition probability to the next location. The flight behavior of some birds or fruit flies has demonstrated the typical characteristics of Lévy flights, which are a form of flight that manifest power law-like characteristics. In this case, the landscape is typically explored by using a series of straight flight paths punctuated by sudden turns. Such behavior has been applied for optimization and optimal search with promising results.

The CS heuristic can be summarized in three idealized rules:

- Each cuckoo lays one egg at a time and dumps it in a randomly chosen nest.
- The best nests with high-quality eggs will be carried over to the next generations.
- The number of available host nests is fixed, and the egg laid by a cuckoo is discovered by the host bird with a probability $p_a \in (0,1)$. In this case, the host bird can either get rid of the egg or simply abandon the nest and build a completely new nest.

When generating new solution $x_i^{(t+1)}$ for, say, a cuckoo *i*, a Lévy flight is performed as

$$x_i^{(t+1)} = x_i^t + \alpha \oplus \text{Lévy}\ (\lambda). \tag{1}$$

where $\alpha > 0$ is the step size which should be related to the scales of the problem of interests. In most cases, $\alpha = 1$ is used. This equation is the stochastic equation for random walk. In general, a random walk is a Markov chain whose next location depends only on the current location and the transition probability. The product $\oplus$

means entrywise multiplications. The Lévy flight essentially provides a random walk while the random step length is drawn from a Lévy distribution

$$Lévy \sim \mu = t^{-\lambda}, \ (1 < \lambda \leq 3), \tag{2}$$

which has an infinite variance with an infinite mean. Here the steps essentially form a random walk process with a power law step length distribution with a heavy tail. The algorithm can also be extended to more complicated cases where each nest contains multiple eggs (a set of solutions). The algorithm can be summarized as in the following pseudo code:

1. begin
2. The objective function $f(x)$, $x = (x_1, \ldots, x_d)'$;
3. Generate an initial population of n host nests (solution vectors), namely x_i ($i = 1, 2, ..., n$);
4. while ($t <$ Max iterations) and (termination condition not achieved)
5. Generate a new solution vector x_{new} via Lévy flight and evaluate its fitness, say F_{new};
6. Randomly select a vector (say, x_j) from the current population and compare the function values $f(x_j)$ and $f(x_{new})$;
7. if ($f(x_{new}) < f(x_j)$),
8. replace x_j by x_{new};
9. end if
10. A fraction (p_a) of the worse nests are abandoned and new nests are generated;
11. Keep the best solutions (or nests with quality solutions);
12. Rank the solutions and find the current best solution vector;
13. end while
14. Post process results and visualization.
15. end

- **Mantegna's algorithm**

Mantegna's algorithm ([6]) produces random numbers according to a symmetric Lévy stable distribution. It was developed by R. Mantegna. The algorithm needs the distribution parameters $\alpha \in [0.3, 1.99]$, $c > 0$, and the number of iterations, n. It also requires the number of points to be generated. When not specified, it generates only one point. If an input parameter will be outside the range, an error message will be displayed and the output contains an array of NaNs (Not a Number). The algorithm is described in the following steps:

$$v = \frac{x}{|y|^{1/\alpha}}, \tag{3}$$

where x and y are normally distributed stochastic variables and

$$\sigma_x(\alpha)=\left(\frac{\Gamma(\alpha+1)\sin\left(\frac{\pi\alpha}{2}\right)}{\Gamma\left(\frac{\alpha+1}{2}\right)\alpha 2^{(\alpha-1)/2}}\right)^{\frac{1}{\alpha}},\quad \sigma_y=1. \tag{4}$$

The resulting distribution has the same behavior as a Lévy distribution for large values of the random variable $(|v|>>0)$. Using the nonlinear transformation

$$w=\left((K(\alpha)-1)e^{-|v|/C(\alpha)}+1\right)v, \tag{5}$$

the sum $z_{cn}=\frac{1}{n^{1/\alpha}}\sum_1^n w_k$ quickly converges to a Lévy stable distribution. The convergence is assured by the central limit theorem. The value of $K(\alpha)$ can be obtained as

$$K(\alpha)=\frac{\alpha\,\Gamma\left(\frac{\alpha+1}{2\alpha}\right)}{\Gamma\left(\frac{1}{\alpha}\right)}\left(\frac{\alpha\,\Gamma\left(\frac{\alpha+1}{2}\right)}{\Gamma(\alpha+1)\sin\left(\frac{\pi\alpha}{2}\right)}\right)^{\frac{1}{\alpha}}. \tag{6}$$

Also, $C(\alpha)$ is the result of a polynomial fit to the values obtained by resolving the following integral equation:

$$\frac{1}{\pi\sigma_x}\int_0^{\alpha} q^{1/\alpha}\exp\left(-\frac{q}{2}-\frac{q^{2/\alpha}C(\alpha)}{2\sigma_x^2}\right)dq = \frac{1}{\pi}\int_0^{\alpha}\cos\left(\left(\frac{K(\alpha)-1}{e}+1\right)C(\alpha)\right)\exp\left(-q^{\alpha}\right)dq \tag{7}$$

The required random variable is given by $z=C^{1/\alpha}z_{cn}$.

- **Simplified version of the algorithm**

Mantegna's algorithm uses two normally distributed stochastic random variables to generate a third random variable which has the same behavior as a Lévy distribution for large values of the random variable. Further it applies a nonlinear transformation to let it quickly converge to a Lévy stable distribution. However, the difference between the Mantegna's algorithm and its simplified version used by Yang and Deb ([13]) as a part of cuckoo search algorithm is that the simplified version does not apply the aforesaid nonlinear transformation to generate Lévy flights. It uses the entry-wise multiplication of the random number so generated and the distance between the current solution and the best solution obtained so far (which look similar to the Global best term in PSO) as a transition

probability to move from the current location to the next location to generate a Markov chain of solution vectors. However, PSO also uses the concept of Local best. Implementation of the algorithm is very efficient with the use of Matlab's vector capability, which significantly reduces the running time. The algorithm starts with taking one by one solution from the initial population and then replacing it by a new vector generated using the steps described below:

$$stepsize = 0.01 \cdot v \cdot (s - current\ best), \qquad (8)$$
$$new_{soln} = old_{soln} + stepsize \cdot z,$$

where v is the same as in Mantegna's algorithm above with σ_x calculated for $\alpha = 3/2$, while z is again a normally distributed stochastic variable.

2.2. Particle Swarm Optimization (PSO)

PSO has been originally proposed by Kennedy and Eberhart ([4], 1995). It is behaviorally inspired and belongs to Evolutionary Computation, whose main purpose is the emergence of complex behaviors from simple rules. In the specific case of PSO, the strategy of searching the problem hyperspace for optimum was developed out of attempts to model the social behavior of bird flocking or fish schooling.

PSO consists of a swarm of particles. Each particle resides at a position in the search space. The fitness of each particle represents the quality of its position. Initially, the PSO algorithm chooses candidate solutions randomly within the search space. The particles fly over the search space with a certain velocity. The velocity (both direction and speed) of each particle is influenced by its own best position found so far and the best solution that was found so far by its neighbors. Eventually the swarm will converge to optimal positions.

Let $i \in \{1, \ldots, N\}$, $x_i \in \Re^n$ and $v_i \in \Re^n$ be a particle, its position and its velocity, respectively. Now, consider a fitness function $f : \Re^n \to \Re$. Candidate solutions x_i are initially placed at random positions in the search-space and moving in randomly defined directions. The direction of a particle is then gradually changed to move in the direction of the best found positions of itself and its peers, searching in their vicinity and potentially discovering better positions.

The pseudo-code of PSO is given below:

1. Initialize all particles i with random positions in the search space: $x_i^0 \sim U(b_{lo}, b_{up})$, where b_{lo} and b_{up} are the lower and upper boundaries of the search-space.
2. Set each particle's best known position to its initial position: $pBest\ _i^0 = x_i^0$.
3. Initialize each particle's velocity to random values: $v_i^0 \sim U(-d, d)$, where $d = |b_{up} - b_{lo}|$.

4. Set the initial swarm's best known position $gBest^0$ to the $pBest_i^0$ for which $f(pBest_i^0)$ is lowest.
5. **repeat**
6. **for** all Particle i in the swarm do
7. Pick two random numbers: ε_p, $\varepsilon_g \sim U(0, 1)$.
8. Update the particle's velocity:

$$v_i^{t+1} = w \cdot v_i^t + c_p \cdot \varepsilon_p \cdot (pBest_i^t - x_i^t) + c_g \cdot \varepsilon_g \cdot (gBest^t - x_i^t) \tag{9}$$

where w is a parameter, called inertia weigth, c_p is the so-called self adjustment coefficient, c_g is the so-called social adjustment coefficient, x_i^t is the current position of particle i at iteration t, $pBest_i^t$ is the best position in the current neighborhood, and *gBest* is the best position so far.

9. Compute the particle's new position:

$$x_i^{t+1} = x_i^t + v_i^{t+1}. \tag{10}$$

10. **if** $f\left(x_i^{t+1}\right) < f\left(pBest_i^t\right)$ then
11. Update the particle's best known position:

$$pBest_i^{t+1} = \left(x_i^{t+1}\right). \tag{11}$$

12. **end if**
13. **if** $f\left(pBest_i^{t+1}\right) < f\left(gBest^t\right)$ then
14. Update the swarm's best known position:

$$gBest^{t+1} = pBest_i^{t+1} \tag{12}$$

15. **end if**
16. **end for**
17. **until** termination criterion is met
18. **return** the best known position: $gBest$.

The first term of (9), $w \cdot v_i^t$, is the inertia component, responsible for keeping the particle moving in the same direction it was originally heading. The role of the coefficient w is either to damp the particle's inertia or to accelerate the particle in its original direction. Generally, lower values of w speed up the convergence of the swarm to optima, and higher values of w encourage exploration of the entire search space.

2.3. Gravitational Search Algorithm (GSA)

Gravitational search algorithm (GSA) was originally proposed by Rashedi et al. ([9], 2009). In GSA, all particles are viewed as objects with masses. Based on the Newton's law of universal gravitation, the objects attract each other by the gravity force, and the force makes all of them move towards the ones with heavier masses. Each mass has four characteristics: position, inertial mass, active gravitational mass, and passive gravitational mass. The first one corresponds to a solution of the problem, while the other three are determined by fitness function.

Let us consider a system with N masses (agents), where the ith mass's position is defined as follows:

$$X_i = \left(x_i^1, \ldots, x_i^d, \ldots, x_i^n\right), \quad i = 1, 2, \ldots, N. \tag{13}$$

The gravitational force acting on mass i from mass j at a specific time t is defined as follows:

$$F_{ij}^d(t) = G(t)\frac{M_{pi}(t)\cdot M_{aj}(t)}{R_{ij}(t)+\varepsilon}\left(x_j^d(t) - x_i^d(t)\right), \tag{14}$$

where M_{aj} is the active gravitational mass related to agent j, M_{pi} is the passive gravitational mass related to agent i, $G(t) = G_0 \cdot e^{-\alpha \cdot iter/maxiter}$ is a gravitational constant that is diminishing with each iteration, ε is a small constant, and $R_{ij}(t) = \left\|X_i(t), X_j(t)\right\|_2$ is the Euclidian distance between two agents i and j.

For the purpose of computing the acceleration of an agent i, total forces (related to each direction d at time t) can be defined by

$$F_i^d(t) = \sum_{j=1,\, j\neq i}^{N} \varepsilon_j \, F_{ij}^d(t), \quad \varepsilon_j \sim U(0,1). \tag{15}$$

Alternatively, to improve the performance of GSA by controlling exploration and exploitation, only the group *Kbest* of heavier agents is allowed to attract the others, where *Kbest* is decreasing over time.

$$F_i^d(t) = \sum_{j\in Kbest,\, j\neq i}^{N} \varepsilon_j \, F_{ij}^d(t), \quad \varepsilon_j \sim U(0,1). \tag{16}$$

Thus, by lapse of iterations, exploration is fading out and exploitation is fading in.

Given the inertial mass M_{ii} of the ith agent, we can now define the acceleration of the agent i, at time t, in the dth direction:

$$a_i^d = \frac{F_i^d(t)}{M_{ii}(t)}. \tag{17}$$

The *i*th agent's next velocity and position can then be computed as:

$$v_i^d(t+1) = \eta_i \cdot v_i^d(t) + a_i^d, \quad \eta_i \sim U(0,1), \tag{18}$$

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1). \tag{19}$$

Finally, after computing current population's fitness, the gravitational and inertial masses can be updated as follows:

$$m_i(t) = \frac{fit_i(t) - worst(t)}{best(t) - worst(t)}, \qquad M_i(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)}. \tag{20}$$

where $fit_i(t)$ is the fitness value of the agent i at time t; $best(t)$ is the strongest agent at time t, and $worst(t)$ is the weakest agent at time t; $best(t)$ and $worst(t)$ are calculated as:

For a minimization problem:
$$\begin{cases} best(t) = \min\limits_{j \in \{1, \ldots, N\}} fit_j(t), \\ worst(t) = \max\limits_{j \in \{1, \ldots, N\}} fit_j(t). \end{cases} \tag{21}$$

For a maximization problem:
$$\begin{cases} best(t) = \max\limits_{j \in \{1, \ldots, N\}} fit_j(t), \\ worst(t) = \min\limits_{j \in \{1, \ldots, N\}} fit_j(t). \end{cases} \tag{22}$$

The steps of implementing GSA can be summarized as follows:

1. Generate the initial population.
2. Evaluate the fitness for all agents.
3. Update the parameters $G(t)$, $best(t)$ and $worst(t)$.
4. Calculate the gravitational and inertial masses $m_i(t)$ and $M_i(t)$ and the total forces $F_i^d(t)$ in different directions , for $i = 1, 2, \ldots, N$.
5. Update the velocities v_i^d and the positions x_i^d .
6. Repeat steps 2 to 5 until the stop criterion is reached. If a specified termination criterion is satisfied, stop and return the best solution.

2.4. The PSO-GSA Hybrid Algorithm

Hybridization itself is an evolutionary metaheuristic approach that mainly depends upon the role of the parameters in terms of controlling the exploration and exploitation capabilities. In principle, we can exploit synergically the mechanisms of control from two algorithms in order to form a hybrid with combined capabilities. This may be more likely to produce better algorithms. The critical parameters in PSO are *pBest*, whose role is to implement the exploration ability, and *gBest*, whose role is to implement the exploitation ability. The critical

parameters in GSA are G_0 and α, which determine the values of $G(t)$, i.e., $G(t) = G_0 \cdot e^{-\alpha \cdot iter/maxiter}$. They allow the fine tuning of the exploitation capability in the first stage and a slow movement of the heavier agents in the last stage. Numerical experiments have shown that PSO performs better in exploitation, whereas GSA performs better in exploration. However, the latter suffers from slow searching speed in the last iterations. A new hybrid algorithm, called PSO-GSA, has been developed by combining the mechanism of these two algorithms and the functionality of their parameters. It was recently proposed by Mirjalili et al. ([5], 2010) and has been tested on twenty-three benchmark functions in order to prove its higher performance compared to standard PSO and GSA. The results shown that that PSO-GSA outperforms both PSO and GSA in most cases of function minimization and that its convergence speed is also faster.

The main difference in PSO-GSA is the way of defining the equation for updating the velocity $V_i\ (t)$:

$$V_i\ (t+1) = w \cdot V_i\ (t) + c_1' \cdot \xi_1 \cdot ac_i(t) + c_2' \cdot \xi_2 \cdot (gBest - X_i\ (t)), \tag{23}$$

where c_1' and c_2' are two adjustable parameters, w is a weighting coefficient, ξ_1, $\xi_2 \sim U(0,1)$ are random numbers, $ac_i(t)$ is the acceleration of agent i at iteration t, and *gbest* is the best solution so far.

The way of updating the agent positions is unchanged:

$$X_i(t+1) = X_i(t) + V_i(t+1). \tag{24}$$

By adjusting the parameters c_1' and c_2' via the updating procedure, PSO-GSA has a better ability to balance the global search and local search. The agents near good solutions try to attract the other agents which are exploring the search space. When all agents are near a good solution, they move very slowly. By using a memory to store the best solution (*gBest*) found so far, PSO-GSA can exploit this information, which is accessible anytime to each agent. Thus the agents can observe the best solution and can tend toward it.

2.5. Many Optimizing Liaisons (MOL)

Here, we consider a metaheuristic derived from PSO, by simplifying its mathematical description in the following way: the particle's own previous best known position *pBest* is eliminated from equation (9), by setting c_p = 0. The velocity update formula now becomes:

$$v_i^{t+1} = w \cdot v_i^t + c_g \cdot \varepsilon_g \cdot (gBest^{\,t} - x_i^t), \tag{25}$$

where ω is still the inertia weight, and $\varepsilon_g \sim U(0,\ 1)$ is a stochastic variable weighted by the user-defined behavioral parameter c_g. The particle's current

position is still denoted by x_i^t and updated as in the PSO method, and the entire swarm's best known position is known as *gBest* as well. The algorithm is also identical to that of the PSO, with the exception that it too can be simplified somewhat by randomly choosing the particle to update, instead of iterating over the entire swarm. This simplified PSO is called *Many Optimizing Liaisons* (MOL) to make it easy to distinguish from the original PSO ([8]).

3. EXPERIMENTS WITH TRAINING CLASSIFIERS

3.1. NN Architecture, Fitness Function and the Encoded Strategy

The architecture of the evolved NN is determined by its topological structure and can be described as a directed graph in which each node performs a transfer function, typically a sigmoid:

$$f(s_j) = \frac{1}{1 + e^{-\left(\sum_{i=1}^{n} w_{ij} \cdot x_i - \theta_j\right)}}, \quad j = 1, 2, \ldots, h, \tag{26}$$

where $s_j = \sum_{i=1}^{n} w_{ij} \cdot x_i - \theta_j$, n is the number of the input nodes, w_{ij} is the connection weight from the *i*th node in the input layer to the *j*th node in the hidden layer, θ_j is the bias (threshold) of the *j*th hidden node, and x_i is the *i*th input.

The final output of the NN can be defined as follows:

$$o_k = \sum_{j=1}^{h} w_{kj} \cdot f(s_j) - \theta_k, \quad k = 1, 2, \ldots, m, \tag{27}$$

where w_{kj} is the connection weight from the *j*th hidden node to the *k*th output node and θ_k is the bias (threshold) of the *k*th output node. The architecture of a NN with 2 inputs, 2 outputs and 3 hidden nodes is shown in Figure 1.

Evolving NN with search heuristics consists of using that heuristic to find the parameters (weights and biases) of the NN as a solution of an optimization problem.

As an encoding strategy we use a connection-based direct encoding of the NN parameters, such as the weights and biases, which are passed, as candidate solutions, to the fitting (objective) function of the population-based optimization algorithm.

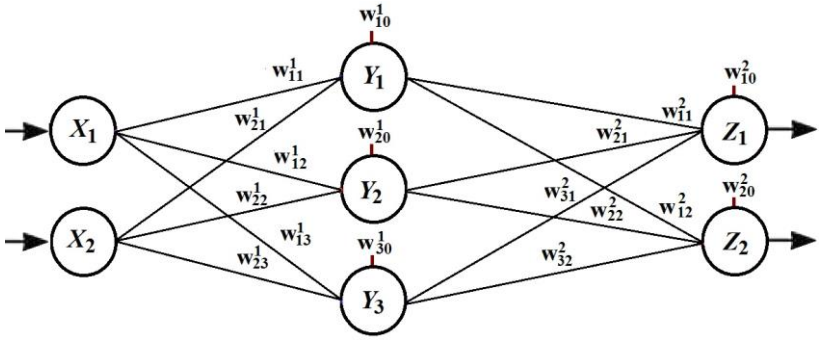

Figure 1. The architecture of a NN with 2 inputs, 2 outputs and 3 hidden nodes

The fitness function is defined in terms of the Mean Square Error (MSE) of the NN. Let us denote by q the number of training samples, by d_i^k the desired output of the ith input unit when the kth training sample is used, and by o_i^k the actual output of the ith input unit when the kth training sample is used. Then:

$$MSE = \sum_{k=1}^{q} \frac{E_k^2}{q}, \text{ where } E_k^2 = \sum_{i=1}^{m} \left(o_i^k - d_i^k\right)^2 . \tag{28}$$

3.2. Predicting Bankruptcy: Experimental Setup and Results

We next discuss the experimental setup proposed in this section. Our purpose is to compare the performances of CS, PSO, GSA PSO-GSA and MOL metaheuristics, when using to evolve neural network classifiers for bankruptcy prediction. This is a hard classification problem, as data are high-dimensional, non-Gaussian, and exceptions are common.

A sample of 130 Romanian companies has been drawn from those listed on Bucharest Stock Exchange (BSE), with the additional restriction of having a turnover higher than one million EURO. Financial results for the selected companies were collected from the 2013 year-end balance sheet and profit and loss account. As predictors, a number of 16 financial ratios have been used in our models. The classification task consists of building classification models from a sample of labeled examples and is based on the search for an optimal decision rule which best discriminates between the groups in the sample.

When evaluating the predictive performance of one or more models, one of the core principles is that out-of-sample data is used to test the accuracy. The validation method we used in our experiments is the holdout method. The data set has initially been split into two subsets; about 60% of the data have been used for training and 40% for testing.

In binary classification, the accuracy is a statistical measure of how well a classifier correctly identifies if an object belongs to one of two groups. However, accuracy is not a reliable metric for the real performance of a classifier, because it

will yield misleading results if the data set is unbalanced. Unfortunately, this is the case with our dataset, where the number of samples in the two classes varies greatly: 104 solvent firms and 26 insolvent ones. Thus, a more detailed analysis than mere proportion of correct guesses (accuracy) is needed. Actually, the performance of the competing classifiers was evaluated using the Confusion Matrix and the Receiver Operating Characteristic (ROC) analysis.

The NN classifiers evolved with CS, PSO, GSA, PSO-GSA and MOL have similar architectures: 16 inputs, 2 outputs (binary response) and 15 hidden nodes. The results for all search heuristics used to evolve NN classifiers for our application at hand (bankruptcy prediction) are summarized in table 1. Figure 2 shows the learning performance of using PSO-GSA to evolve the NN. Figures 3-4 show the confusion matrices and ROC curves for training and test datasets. Similar graphical representations can be obtained for the other metaheuristics.

Table 1. In-sample and out-of-sample average classification error rates

	PSO-GSA	CS	MOL	PSO	GSA
In-sample classification rate	98.7 %	96.2 %	93.6 %	92.3 %	91.0 %
Out-of-sample classification rate	90.4 %	88.5 %	88.5 %	86.5 %	84.6 %

As we expected, the in-sample classification rates are better than the out-of-sample classification rates for all algorithms. The hybrid metaheuristic PSO-GSA outperforms all the standalone metaheuristics, either in case of already seen (training) data, or in case of unseen (test) data, showing that the hybridization is an effective approach. As for the standalone metaheuristics, the best-performing are CS and MOL.

Further experiments are intended in order to evaluate repeatedly the performance of the three algorithms in terms of average classification rates, or to apply them on other databases.

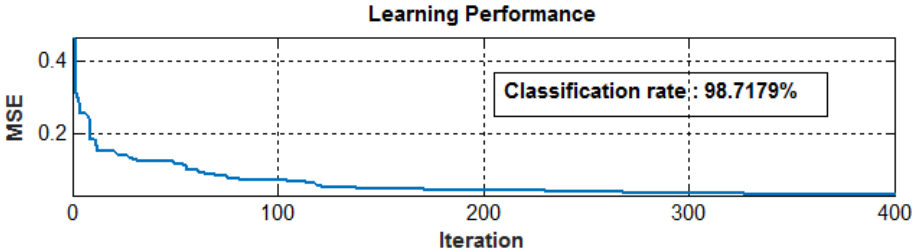

Figure 2. NN evolved with PSO-GSA: Learning performance

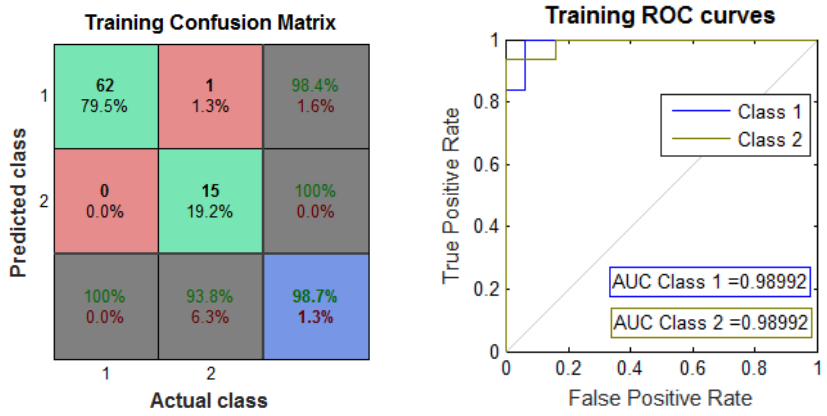

Figure 3. NN evolved with PSO-GSA: Training confusion matrix and ROC curves

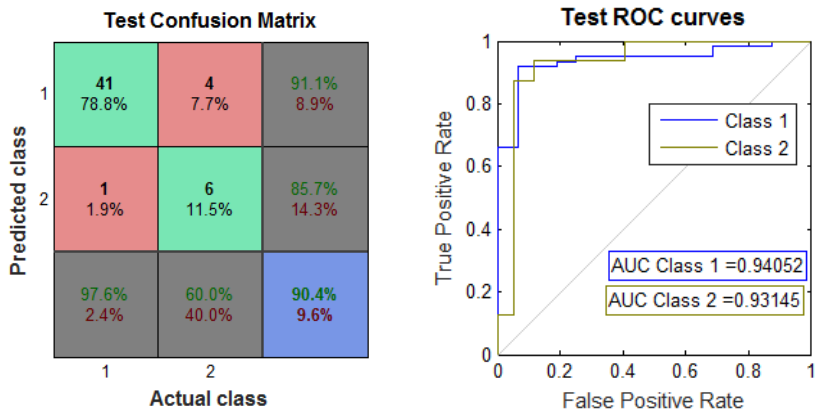

Figure 4. NN evolved with PSO-GSA: Test confusion matrix and ROC curves

4. EXPERIMENTS WITH TRAINING FUNCTION APPROXIMATORS

4.1. NNARX models

Feedforward Neural Networks offer a straightforward extension to the classical way of modeling time series. Namely, they can use a specific mechanism to deal with temporal information (a series-parallel architecture for one-step ahead prediction with respect to both delayed inputs and outputs, without feedback**)** and can thus extend the linear autoregressive model with exogenous variables (ARX) to the *nonlinear ARX* form:

$$y_t = F\left(y_{t-1}, \ldots, y_{t-na}, X_{t-nk}, \ldots, X_{t-nk-nb}\right) + \varepsilon_t \tag{29}$$

where F is a non-linear function, n_a is the number of past outputs, n_b is the number of past inputs and n_k is the time delay.

Nonlinear neural network ARX (NNARX) models are potentially more powerful than linear ones in that they can model more complex underlying characteristics of time series and theoretically do not have to assume stationarity.

In a NNARX model, y_t is a function of its lagged values y_{t-j} and the lagged values of some exogenous variables, having the role of capturing extraneous influences. For example, extending the NNAR model to a NNARX one, by allowing the LEU/EURO exchange rate to additionally depend on some other exchange rate as an exogenous variable (say, the LEU/USD exchange rate) proved to ameliorate the overall forecasting performance.

In principle, the output of the NNARX model can be considered as an estimate $\hat{y}_t$ of the output y_t of some nonlinear dynamical system and thus it should be feed back in the next stage to the input of the feedforward neural network. However, because the true previous outputs y_{t-j} are available at time t during the training of the network, a series-parallel architecture can be created, in which the true outputs y_{t-j} are used instead of feeding back the estimated outputs $\hat{y}_{t-j}$, as shown in Figure 5. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and the backpropagation or, alternatively, a metaheuristic, can be used statically for training.

We adopted the standard approach for training and testing, that is, to evaluate a model by testing its performance on a validation set consisting of out-of-sample data.

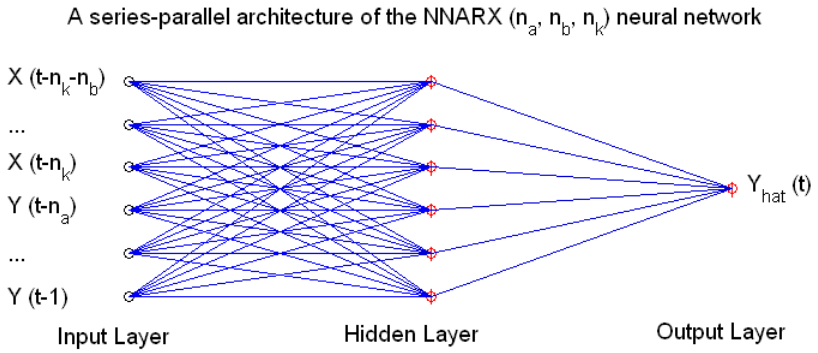

Figure 5. A purely feedforward architecture of the NNARX(n_a, n_b, n_k) neural Net

4.2. Predicting Foreign Exchange Rates: Experimental Setup and Results

Let us consider a NNARX(2, 2, 1) model:

$$y_t = F\left(y_{t-1}, y_{t-2}, x_{t-1}, x_{t-2}\right) + \varepsilon_t, \tag{29}$$

where the Romanian Leu to Euro exchange rate (denoted by y_t) is the endogenous variable, the Romanian Leu to US Dollar exchange rate (denoted by x_t) is the exogenous variable, $n_a = 2$, $n_b = 2$ and $n_k = 1$. The additional information provided by x_t is intended to ameliorate the overall forecasting performance for y_t. The neural network architecture associated with this model is depicted in Figure 4. The number of hidden neurons is 12. The dimension of the search space is 73 (4×12+12=60 weights, 12+1=13 biases).

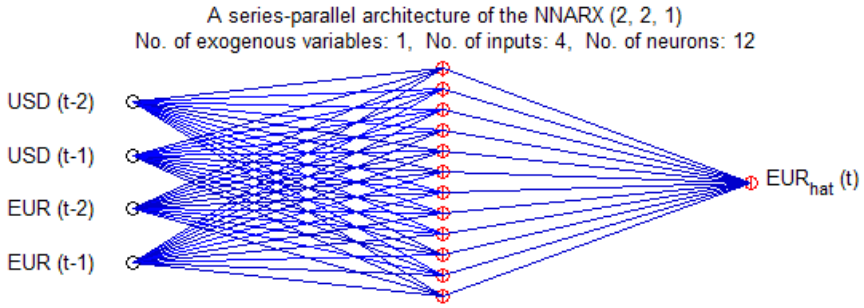

Figure 6. The NN architecture of the NNARX(2, 2, 1) model

The fitness function used to evolve the NNARX models over the training data, by means of each metaheuristic in turn, is defined as the Mean Square Error (MSE). MSE is also used to measure the predictive performances of the evolved models, for all datasets: training, validation and test. The results of our experiments are listed in table 2. The last 3 metaheuristics are hybridizations of CS, PSO and GSA with the Matlab local optimizer FMINCON.

Table 2

Metaheuristics	Rank	MSE-training	MSE-validation	MSE-test
PSO-GSA	1	0.00024543	0.0000645492	0.000192541
CS	3	0.00025360	0.0000716124	0.000070092
MOL	4	0.00026799	0.0000730520	0.000134811
PSO	6	0.00035111	0.0000795555	0.000085198
GSA	8	0.00053956	0.0002893300	0.000286416
CS-FMINCON	2	0.00024790	0.0000654016	0.000091281
PSO-FMINCON	5	0.00027457	0.0000713060	0.000140976
GSA-FMINCON	7	0.00036225	0.0001269550	0.000124204

It is worth to note the important role of hybridization in improving the predictive performance of the evolved models. The best-performing optimizers are two hybrid metaheuristics (PSO-GSA and CS-FMINCON) that are more effective while in tandem than working alone.

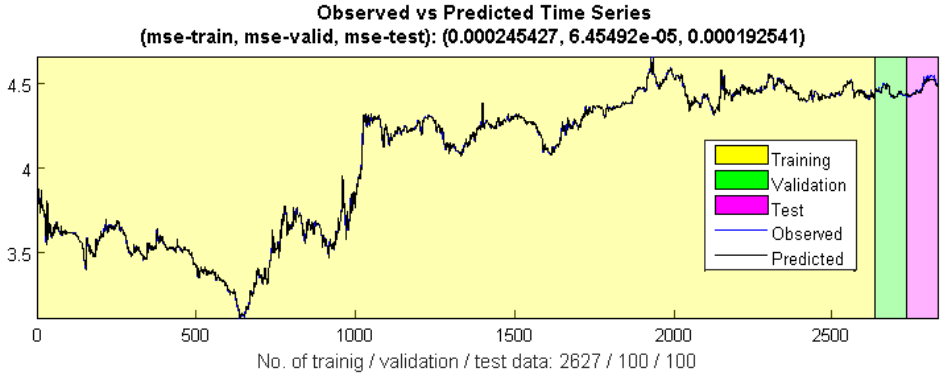

Figure 7. The PSO-GSA hybridization (PSO-GSA)

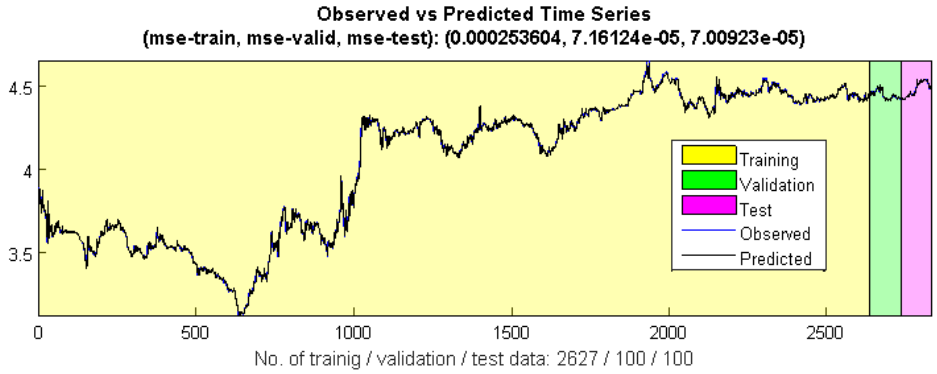

Figure 8. Cuckoo Search (CS)

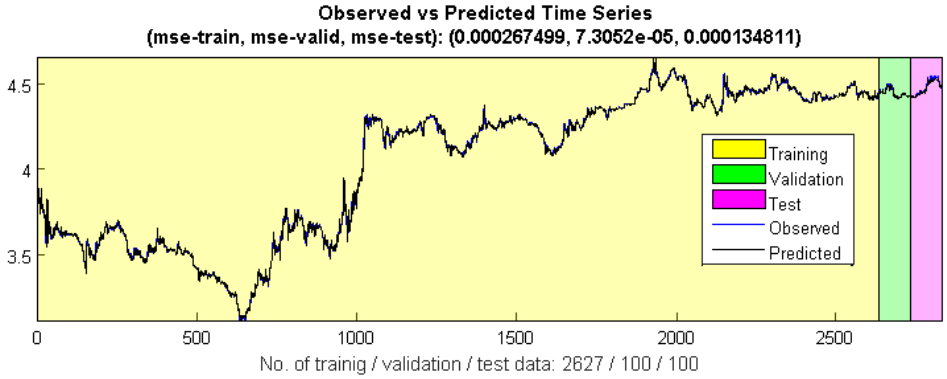

Figure 9. Many Optimizing Liaisons (MOL)

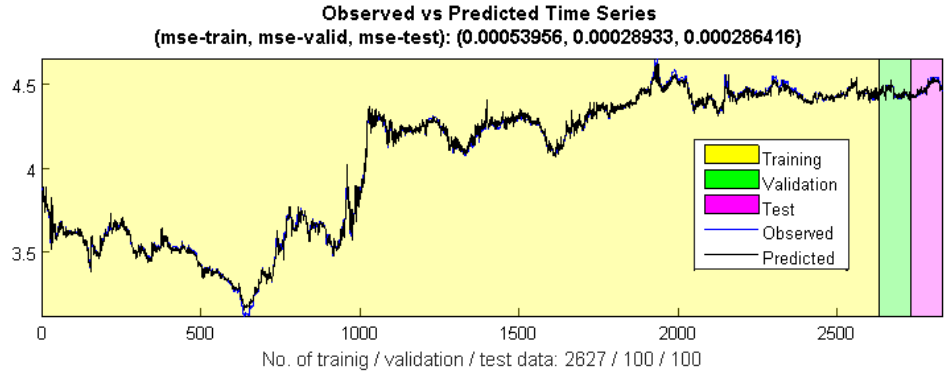

Figure 10. Gravitational Search Algorithm (GSA)

As for the standalone metaheuristics, the best-performing are, by far, Cuckoo Search and Many Optimizing Liaisons.

A final note is about the remarkable generalization capability of all metaheuristics: the MSE for validation and test datasets are surprisingly good, even better than the MSE for training dataset. This emphasizes once again the very nature of metaheuristics as global optimizers.

5. CONCLUSIONS

The aim of this paper was to compare the performances of several nature-inspired metaheuristics, when using alone, or synergically, as hybrid methods, for evolving Neural Network based classifiers and function approximators. Founded on a new computational paradigm, the optimizers in this category have been proposed in an attempt to overcome the weakness of gradient-based techniques. Backpropagation, the most popular technique for training NNs, is also based on gradient descent and thus suffers from the same drawbacks as the classical optimizers, including a low convergence rate, instability, the differentiability condition, the risk of being trapped in a local minimum and the possibility of overshooting the minimum of the error surface. Metaheuristics, instead, are gradient-free, have a high convergence rate and are able to perform notably well in complex task, such as training neural networks that generate complex error surfaces, with multiple local minima.

Hybridization has often proven to be a successful approach in many cases where, by combining the mechanism of two algorithms and the functionality of their parameters, we are able to find new ways of controlling the exploration and exploitation capabilities of the newly generated hybrid. The results reported here may be seen as another step in validating approaches of this kind.

REFERENCES

[1] **Eiben, A.E., Schippers, C.A. (1998),** ***On Evolutionary Exploration and Exploitation****; Fundamenta Informaticate*, vol. 35, no. 1-4, 35-50;

[2] **Georgescu, V. (2015),** ***Using Genetic Algorithms to Evolve a Type-2 Fuzzy Logic System for Predicting Bankruptcy*** ; Advances in Intelligent Systems and Computing, Vol. 377, pp 359-369, *Springer;*

[3] **Georgescu, V. (2010*), Robustly Forecasting the Bucharest Stock Exchange BET Index through a Novel Computational Intelligence Approach***; *Economic Computation and Economic Cybernetics Studies and Research, ASE Publishing;* 44 (3), 23-42;

[4] **Kennedy, J., Eberhart, R.C. (1995),** ***Particle Swarm Optimization***, in Proceedings of IEEE international conference on neural networks, vol. 4, 1942–1948;

[5] **Mirjalili, S., Mohd Hashim, S.Z. (2010*), A New Hybrid PSOGSA Algorithm for Function Optimization***, in: International Conference on Computer and Information Application (ICCIA 2010), 374-377;

[6] **Mantegna, R. (1994),** ***Fast, Accurate algorithm for Numerical Simulation of Lévy Stable Stochastic Processes***; *Physical Review* E, Vol. 49, No. 5, 4677-4683;

[7] **Pearl, J. (1983),** ***Heuristics: Intelligent Search Strategies for Computer Problem Solving****; Addison-Wesley;*

[8] **Pedersen M.E.H., Chipperfield A.J. (2010),** ***Simplifying Particle Swarm Optimization****;* Applied Soft Computing; Volume 10, Issue 2, 618–628;

[9] **Rashedi, E., Nezamabadi, S., Saryazdi, S. (2009),** ***GSA: A Gravitational Search Algorithm****; Information Sciences*, vol. 179, no. 13, 2232- 2248;

[10] **Rodrigues, D., Pereira, L.A.M., Souza, A.N., Ramos, C.C., Xin-She Yan (2013),** ***Binary Cuckoo Search: A Binary Cuckoo Search Algorithm For Feature Selection***; IEEE International Symposium on Circuits and Systems (ISCAS), 465 - 468 ;

[11] **Shi, Y., Eberhart, R.C. (1998),** ***A Modified Particle Swarm Optimiser***, in IEEE International Conference on Evolutionary Computation, Anchorage, Alaska;

[12] **Walton, S., Hassan, O., Morgan, K., Brown, M.R. (2011),** ***Modified Cuckoo Search: A New Gradient Free Optimization Algorithm***. Chaos, Solitons & Fractals, 44(9), 710–718;

[13] **Yang, X.-S., Deb S. (2009),** ***Cuckoo Search via Lévy Flights***, in Proceedings of World Congress on Nature & Biologically Inspired Computing (NaBIC 2009), India, *IEEE publications*, USA, 210-214.