

Chapter 2

Evolutionary Computing and Type-2 Fuzzy Neural Networks

2.1 Evolutionary Computing Methods

Evolutionary computing involves stochastic search and continuous optimization methods that are inspired by biological mechanisms and systems (Crosby 1973; Eiben and Smith 2003). These computing methods inherit the principles of development and progress from the natural processes and phenomena such as evolution, reproduction (or generation), selection, survival, grouped and distributed behavior, chance, inheritance, crossover (or recombination), mutation, fitness (or health) and so on. Evolutionary computing methods emulate the laws of natural evolution such as “a stronger (healthier or fitter) organism has more chances to survive than a weaker one”, “an organism or a pair can generate a new offspring with a probability”, “an offspring takes over some of properties of their parents”, “an offspring very rarely but may have some properties that differ it from their parents”, “population size cannot grow infinitely”, etc. Some less natural laws can exist as well: “the best organism will never die”.

Very often evolutionary computing based methods are also named population based. This is because the notion of “population (of individuals)” forms the basis and exists in all such methods whereas the type (i.e. its design and set of properties) of individuals and of the population as well as the processing algorithms to evolve the population may vary in a wide range. In all evolutionary computing methods, every individual is attached a numerical value reflecting its fitness (quality, healthiness). There should also be provided a way to derive the individual’s fitness degree from its properties.

Well-known evolutionary computing techniques are: Differential Evolution (Storn and Price 1997; Price et al. 2005; Feoktistov 2006), Swarm Optimization (Bonabeau et al. 1999; Clerc 2006; Kennedy and Eberhart 1995), Ant Colony Optimization (Dorigo and Stützle 2004), Cultural Algorithms (Reynolds 1994), Harmony Search (Geem et al. 2001; Karahan et al. 2012; Ricart et al. 2011),

Genetic Algorithm (Chiong et al. 2012; Goldberg 1989; Langdon and Poli 2002) and others.

When applying an evolutionary computing based approach for solving optimization problems a candidate solution (i.e. appropriate values of sought-for variables) is represented as an individual in a population and the corresponding value of objective function (possibly normalized) as the individual's fitness degree. Generation of new individuals (and accordingly the candidate solutions), their survival, and overall treatment of the population are governed by the laws of evolution driven by application of multiple so-called *evolutionary forces* (or operators) implemented within a specific evolutionary computing technique. The most important and frequently used evolutionary forces are recombination (crossover), mutation, selection, and elitism. While recombination and mutation creates diversity in the population (and accordingly, in the candidate solutions), selection and elitism increases its quality. Thereby it is implemented a global and continuous optimization process.

In some techniques such as Genetic Algorithms (GA) it may be required some transformation procedure to get the problem's decision or search variables into individuals and back (encoding/decoding). Or more specifically: the procedure to convert the variables' numerical values into instances of the individual's *container class* (i.e. a specific data type with fields representing an individual's properties) and vice versa. In GA they often call such data containers as chromosomes or genes (genomes). Physically, in computers, the chromosomes are represented as long strings of bits. As they express it in genetic algorithms, the phenotype (numeric values of problem's variables) is encoded to produce a genotype (a data container – gene or chromosome).

The assessment of the individuals is done by a function called a fitness function (GA) or by a computational model that allows computation of the individual's fitness degree from its properties (phenotype or genotype). If the algorithm uses data containers such as chromosomes, at some stage of the evolution the best (healthiest, strongest, or fittest) chromosome should have been decoded to retrieve the corresponding values of decision variables (e.g. the sought-for solution).

For optimization problems solved by evolutionary computing methods, the fitness function (sometimes, in such techniques as Differential Evolution – DE – replaced by cost or error function) is produced from the objective function and, possibly, constraints posed on the decision variables.

Genetic Algorithm (GA) is one of the first offered evolutionary computing methods and is still very popular. Figure 2.1 in a very general form illustrates the scheme of GA.

Please notice the force named elitism that we include in the scheme of GA shown in Fig. 2.1. The *elitism* force ensures that at least one of best chromosomes is transferred to the next generation. The *elitism* force is not an absolutely necessary one and does not exist in basic versions of GA. However, it is very useful to guarantee the best ever reached historical solution is never lost.

Figure 2.2 illustrates a possible version of implementation of the Genetic Algorithm.

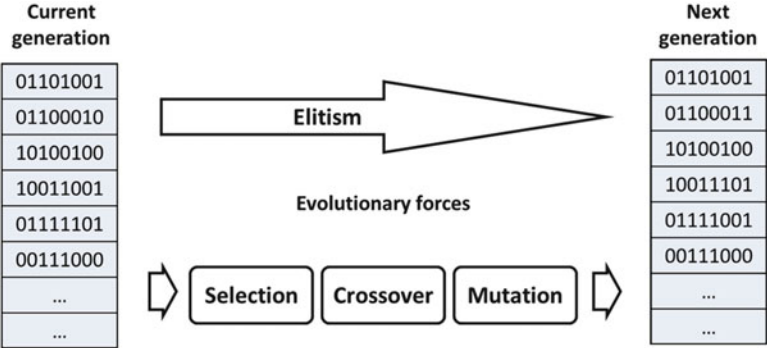


Fig. 2.1 The genetic algorithm

This version of GA requires five user defined parameters: population size (*PopSize*), probability of crossover (*CrossoverProb*), probability of mutation (*MutationProb*), maximum number of generations (*MaxGen*), and desired fitness (*FitnessDesired*). It is assumed that there exist functions: (1) to produce the chromosome from a candidate solution (*get_genotype*), (2) to produce the candidate solution from a chromosome (*get_phenotype*), and (3) to evaluate the quality (fitness degree) of a candidate solution (*fitness*). Also it is assumed that there are operators implementing evolutionary forces (*crossover* and *mutate*). The *crossover* force takes two chromosomes as its arguments to produce a new offspring chromosome, and the *mutate* force alters its single argument chromosome. An *i*-th chromosome is designated *Chromosome[i]*, *i* = 1, ..., *PopSize*. The algorithm stops when the desired fitness threshold (*FitnessDesired*) is reached by one of the chromosomes or the maximum generation number (*MaxGens*) is passed.

As can be seen from the algorithm presented in Fig. 2.2, at steps 3.1–3.4 the bunch of evolutionary forces *elitism*, *selection*, *crossover* (recombination), and *mutation* are applied to the chromosomes in the existing population to create members of the new population. After meeting the stop condition, the sought-for solution is extracted from the chromosome with maximum fitness.

The versions of GA may differ from not only the set but also the type of evolutionary forces (genetic operators) used. Figure 2.3 illustrates various implementations of the crossover and mutation forces.

Let’s now consider an example of optimization using the GA (Fig. 2.4).

Example Find the maximum of the function:

$$z = f(x, y) = 3(1 - x)^2 e^{-x^2 - (y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2 - y^2} - \frac{1}{3} e^{-(x+1)^2 - y^2}$$

Figure 2.5 demonstrates how the initial 20 candidate solutions are distributed and how the population changes after five and ten generations.

GA(PopSize, SelectionProb, MutationProb, MaxGen, FitnessDesired)

- Step 1.** Create and initialize *Population* with *PopSize* random chromosomes: $Ch[i]$, $i=(1,..,PopSize)$. $Gen=0$
- Step 2.** Evaluate fitness for each chromosome in *Population*:
 $Fitness[i]=fitness(get_phenotype(Cromosome[i]))$
- Step 3.** Create new generation *NewPopulation*:
- Step 3.1.** Apply *elitism*: copy the best *Cromosome* $[arg \max_i (Fitness[i])]$ from *Population* to *NewPopulation*.
- Step 3.2.** Select two chromosomes cr_k and cr_l from *Population* with probabilities $\frac{Fitness[k]}{\sum_j Fitness[j]}$ and $\frac{Fitness[l]}{\sum_j Fitness[j]}$, respectively.
- Step 3.3.** Apply crossover force for the selected pare (cr_k, cr_l) with *CrossoverProb* to produce an offspring: $cr_{new} = crossover(cr_k, cr_l)$
- Step 3.4.** Apply mutation force with *Mutation Prob*: $cr_{new} = mutate(cr_{new})$
- Step 3.5.** Copy cr_{new} to *NewPopulation*.
- Step 3.6.** If number of chromosomes in *NewPopulation* $< PopSize$ go to Step3.2.
- Step 4.** Replace *Population* with *NewPopulation*. Increment $Gen=Gen+1$
- Step 5.** Extract the solution:
 $Solution=get_phenotype(Cromosome[arg \max_i (Fitness[i])])$
- Step 6.** Check the termination condition: If $\left(\max_i (Fitness[i]) \geq FitnessDesired \right)$ or $(Gen \geq MaxGens)$ terminate and return *Solution*, else go to Step2
-

Fig. 2.2 Genetic algorithm implementation high level code

Particle swarm optimization (PSO) is another population based stochastic optimization technique inspired by the social behavior of birds (Kennedy and Eberhart 1995, 2001). The algorithm is very simple but powerful. The PSO algorithm is quite similar to genetic algorithms and can be used for similar problems. Because the algorithm allows for parallelization and uses less computational resources than GA, it can efficiently be used to minimize/maximize high dimensional functions and thus as a training method for neural networks.

To understand the algorithm, it is best to imagine a swarm of birds that are searching for food in a defined area. It is assumed that there is only one piece of food in this area. Initially, the birds don't know where the food is, but they know at

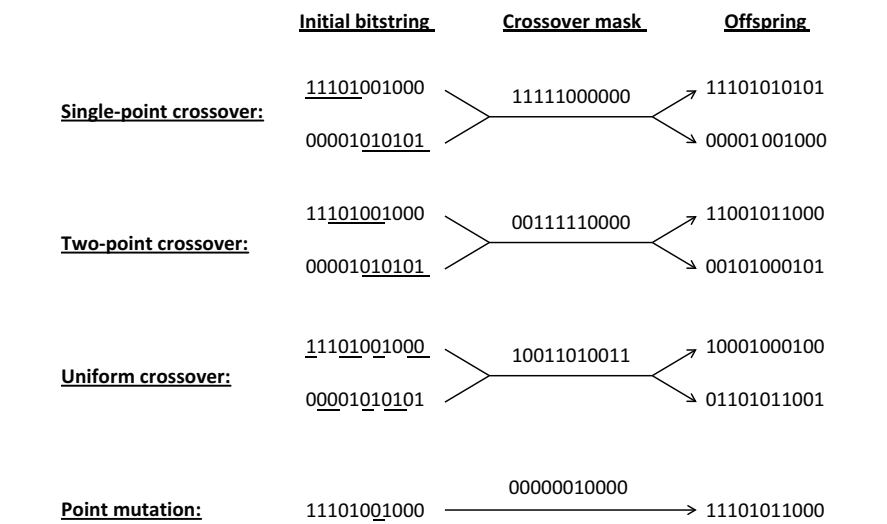


Fig. 2.3 Implementation of operators for GA

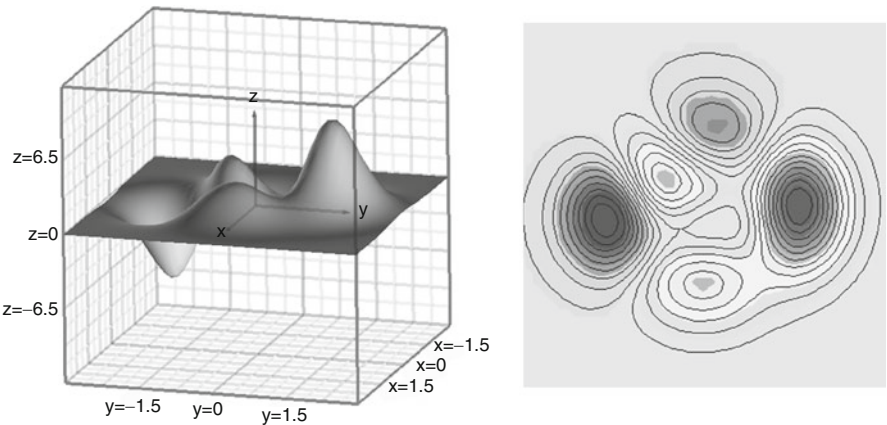


Fig. 2.4 Example function to maximize using GA

each time how far the food is. It is quite natural that in their search the birds will follow the strategy that is nearest to the food.

PSO adopts this behavior and searches the search space for candidate solutions, which are called here particles. Very similar to the GA, each particle has its cost degree (fitness) that is evaluated by a function to be minimized, and each particle has a velocity that directs its flying.

The swarm is initialized by particles at random positions and then each particle flies through the search space by adjusting its velocity (vector **V**) and location (vector **X**) to follow two best candidate solutions found so far: their own best

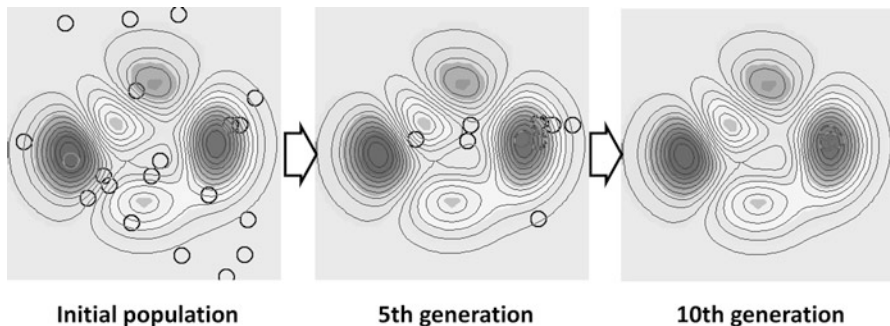


Fig. 2.5 The process of optimization of the example function by GA

PSO(*PopSize*, c_1 , c_2 , *MaxGen*, *MinimumCost*)

- Step 1. Initialize each particle with a random velocity and random position.
 - Step 2. Calculate the cost for each particle. If the current cost is lower than the best value so far, remember this position (*ParticleBest*).
 - Step 3. Choose the particle with the highest fitness (lowest cost) of all particles. The position of this particle is *GlobalBest*.
 - Step 4. Calculate, for each particle, the new velocity and position according to the equations (2.1).
 - Step 5. Go to Step 2 and repeat steps 2-4 until one of the criteria (maximum iterations *Max Genor* minimum cost *Minimum Cost*) is not attained.
-

Fig. 2.6 Basic PSO algorithm

historical location (*ParticleBest*) and the best particle in the swarm (*GlobalBest*). The following equations demonstrate how the adjustments are made:

$$\begin{aligned} \mathbf{V}_{new} &= \mathbf{V} + c_1 r_1 (\mathbf{ParticleBest} - \mathbf{X}) + c_2 r_2 (\mathbf{GlobalBest} - \mathbf{X}) \\ \mathbf{X}_{new} &= \mathbf{X} + \mathbf{V}, \end{aligned} \quad (2.1)$$

where \mathbf{V} is the current velocity, \mathbf{V}_{new} is the new velocity, \mathbf{X} is the current position, \mathbf{X}_{new} is the new position, r_1 and r_2 are random numbers in the interval $[0, 1]$, and c_1 and c_2 are acceleration coefficients: c_1 is the factor that influences the cognitive behavior, i.e. how much the particle will follow its own best solution, and c_2 is the factor for social behavior, i.e. how much the particle will follow the swarm's best solution.

In an optimization problem, the current position will be meant as the vector of sought-for parameter values, i.e. a candidate solution.

The basic algorithm can be written as presented in Fig. 2.6:

The next section will consider another evolutionary optimization technique, which is very efficient and suitable, in our opinion, for application in training of parameters of neural networks – the differential evolution (DE) algorithm.

2.2 Differential Evolution Based Optimization (DEO)

In this section we consider the Differential Evolution (DE) algorithm. This population based algorithm implements global search. Being designed specifically for numerical optimization, it is characterized by good convergence properties in multidimensional search spaces. DE has been successfully applied to solve a wide range of problems such as those found in image classification, clustering, and function optimization. These characteristics of DE make the method also an efficient tool for implementation of neural network training.

2.2.1 DE Algorithm

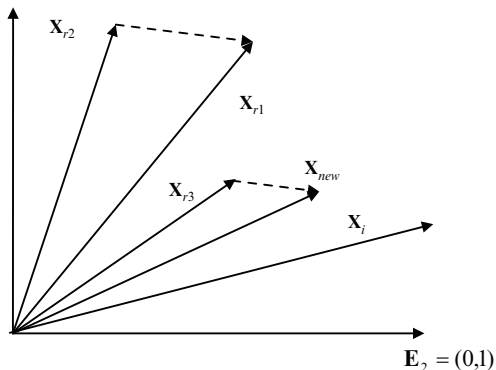
As other stochastic and population-based methods, DE algorithm (Price et al. 2005; Storn and Price 1997; Chakraborty 2008; Feoktistov 2006) uses an initial population of randomly generated individuals and applies to them operations of differential mutation, crossover, and selection. DE considers individuals as vectors in n -dimensional Euclidean space. The population of $PopSize$ ($PopSize \geq 4$) individuals is maintained through consecutive generations. A new vector is generated by mutation, which, in this case is completed by adding a weighted difference vector of two individuals to a third individual as follows: $\mathbf{X}_{new} = (\mathbf{X}_{r1} - \mathbf{X}_{r2})f + \mathbf{X}_{r3}$, where $\mathbf{X}_{r1}, \mathbf{X}_{r2}, \mathbf{X}_{r3}$ ($r1 \neq r2 \neq r3$) are three different individuals randomly picked from the population and f (>0) is the mutation parameter. The mutated vector then undergoes crossover with another vector thus generating a new offspring.

The selection process is realized as follows. If the resulting vector is better (e.g. yields a lower value of the cost function) than the member of the population with an index changing consequently, the newly generated vector will replace the vector with which it was compared in the following generation. Another approach, which we adopted in this research, is to randomly pick an existing vector for realizing crossover.

Figure 2.7 illustrates a process of generation of a new trial solution (vector) \mathbf{X}_{new} from three randomly selected members of the population $\mathbf{X}_{r1}, \mathbf{X}_{r2}, \mathbf{X}_{r3}$. Vector \mathbf{X}_i , $i = 1, \dots, PopSize$, $i \neq r1 \neq r2 \neq r3$ becomes the candidate for replacement by the new vector, if the former is better in terms of the DE cost function. Here, for illustrative purposes, we assume that the solution vectors are of dimension $n = 2$ (i.e. two parameters are to be optimized).

The algorithm itself can be described as presented in Fig. 2.8.

Fig. 2.7 Realization of DE optimization: a two-dimensional case



$\text{DEO}(\text{PopSize}, f, cr, \text{MaxGen}, \text{MinimumCost})$

Step 1. Randomly generate PopSize parameter vectors (from respective parameter spaces (e.g. in the range $[-1, 1]$) and form a population $P = \{\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_{ps}\}$

Step 2. While the termination condition (maximum iterations MaxGen reached or minimum cost MinimumCost attained) is not met generate new parameter sets:

Step 2.1. Choose a next vector \mathbf{X}_i ($i=1, \dots, \text{PopSize}$)

Step 2.2. Choose randomly different 3 vectors from P : $\mathbf{X}_{r1}, \mathbf{X}_{r2}, \mathbf{X}_{r3}$ each of which is different from current \mathbf{X}_i

Step 2.3. Generate trial vector $\mathbf{X}_t = \mathbf{X}_{r1} + f(\mathbf{X}_{r2} - \mathbf{X}_{r3})$

Step 2.4. Generate a new vector from trial vector \mathbf{X}_t . Individual vector parameters of \mathbf{X}_t are inherited with probability cr into the new vector \mathbf{X}_{new} . If \mathbf{X}_{new} evaluates as being a better solution than \mathbf{X}_i , then the current \mathbf{X}_i is replaced in population P by \mathbf{X}_{new}

Next i

Step 3. Select from population P the parameter vector \mathbf{X}_{best} , which is evaluated as the best solution

Step 4. Stop the algorithm

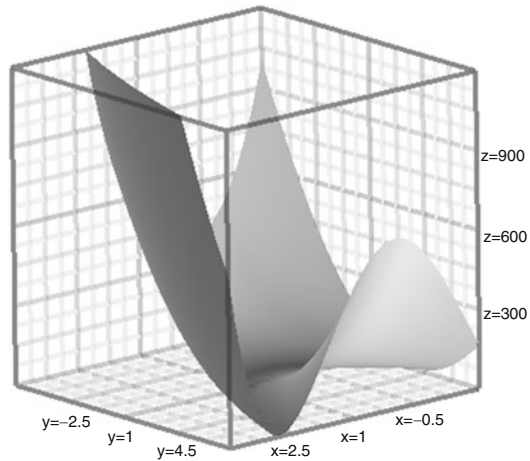
Fig. 2.8 Basic DE algorithm

Usually the mutation rate f is chosen $f \in [0, 2]$. After the crossover of the trial vector and the vector $\mathbf{X}_i = (x_i[1], x_i[2], \dots, x_i[n])$ from the population, at least one of elements (dimensions) of the trial vector $\mathbf{X}_t = (x_t[1], x_t[2], \dots, x_t[n])$ should be transferred to the offspring vector $\mathbf{X}_{new} = (x_{new}[1], x_{new}[2], \dots, x_{new}[n])$. The crossover parameter $cr \in [0, 1]$ affects the mutated (trial) vector as follows:

$$x_{new}[j] = \begin{cases} x_{new}[j], & \text{if } \text{rand}(0, 1) \leq cr \text{ or } \text{rand}(1, n) = j, \\ x_i[j], & \text{otherwise,} \end{cases}$$

where the function $\text{rand}(a, b)$ returns a random value in the range $[a, b]$.

Fig. 2.9 The Rosenbrock function



Differential Evolution based Optimization (DEO) usually implies existence of a function $F(\mathbf{X})$, where \mathbf{X} is vector (x_1, \dots, x_n) , whose values should be minimized by DE algorithm. In DE based neural network training $F(\mathbf{X})$ is replaced by the network error function.

When solving optimization problems a common recommendation is to choose *PopSize* ten times the number of optimization variables (Price et al. 2005), $f = 0.9$, $cr = 1$ or $cr = 0.5$. Some authors suggest ways for choosing optimal values of DE parameters f and cr . For example (Brest et al. 2006) suggests a way for self-adapting of the DE parameters during the optimization.

Let's consider a couple of examples of function optimization using DE.

Example Find the minimum of the function (the Rosenbrock function): $z(x, y) = 100(x^2 - y)^2 + (x - 1)^2$, see Fig. 2.9.

Using the code implementing the DE algorithm, we can see that the algorithm very quickly finds the minimum of this function ($z_{\min} = 0$ reached at $(x_{\min}, y_{\min}) = (1, 1)$) In Fig. 2.10 you can see the average number of calculations of the function to reach the minimum for specified error levels. The experiment has been done with the standard version of DE with the parameters set as *PopSize* = 10, $f = 0.9$, $cr = 1$.

DE is also very effective for multi-parametric function optimization and outperforms most classical and other evolutionary algorithms in finding global minimums for non-smooth and multi-extreme functions. The suggested standard version of the DE algorithm (with *PopSize* set to 3,000) used for minimization of the 50-dimensional Rosenbrock function has used on average about 12 million function evaluations to reach accuracy of 10^{-6} . This performance is comparable with gradient-based methods. However, it should be noted that the well-known coordinate decent, gradient based, and many other classical methods are not global optimizers (the Rosenbrock function is indeed a single-extreme function) and

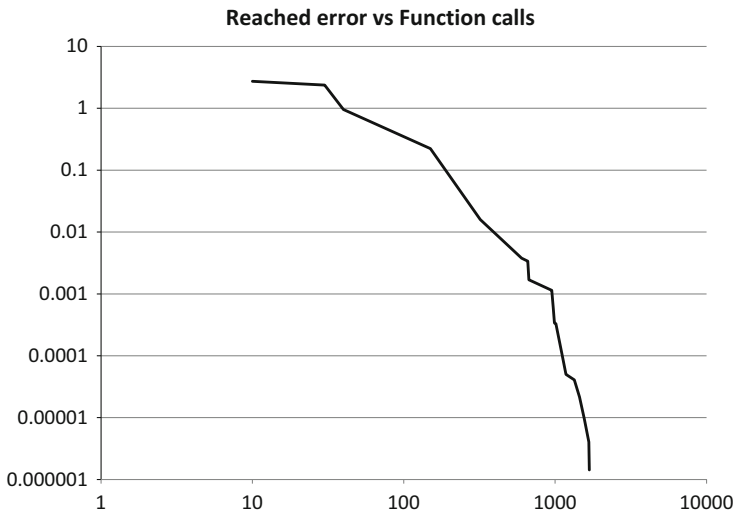


Fig. 2.10 Performance of DE being used for minimization of the Rosenbrock function

pose certain requirements on the function such as smoothness, continuity, differentiability, etc.

Let's consider another example with a more complex function with many local minima.

Example Find the minimum of Griewangk's function (Griewangk 1981):

$$f(\mathbf{X}) = \sum_{i=0}^9 \frac{x_i^2}{4000} - \prod_{i=0}^9 \cos\left(\frac{x_i}{\sqrt{i+1}}\right) + 1, x_i \in [-400, 400].$$

Its global minimum $f(\mathbf{0}) = 0$ is very difficult to find. The standard DE version ($PopSize = 50$) has used on average about 32,000 function evaluations to find the global minimum with the accuracy 10^{-6} .

The Tables 2.2 and 2.3 compares the performance of DE and other methods on a number of benchmark functions presented in Table 2.1 (Vesterstrøm and Thomsen 2004; Yao and Liu 1996).

For DE the following control parameters were set: $PopSize = 100$, $cr = 0.9$, $f = 0.5$. For PSO: $PopSize = 25$, $c_1 = c_2 = 1.8$. For GA: $Popsize = 100$, $MutationProb = 0.9$, $CrossoverProb = 0.7$.

For each problem 30 runs of each algorithm were done and the mean value of function minimum reached and the standard deviation were computed. In Tables 2.2 and 2.3 the best performing algorithms for each problem are marked in bold.

As can be seen from the results of experiments presented above (Vesterstrøm and Thomsen 2004), the performance of DE is outstanding in comparison to the other evolutionary algorithms tested. DE has found the optimum in almost every run.

Table 2.1 Numerical benchmark problems

#	Function	Dimension	Ranges	Minimum
1	$\sum_{i=0}^{n-1} x_i^2$	30/100	$[-5.12, 5.12]$	$F(\mathbf{0}) = 0$
2	$\sum_{i=0}^{n-1} x_i + \prod_{i=0}^{n-1} x_i$	30/100	$[-10, 10]$	$F(\mathbf{0}) = 0$
3	$\sum_{i=0}^{n-1} (\sum_{j=0}^i x_j^2)^2$	30/100	$[-100, 100]$	$F(\mathbf{0}) = 0$
4	$\max x_i , i = 0, \dots, n-1$	30/100	$[-100, 100]$	$F(\mathbf{0}) = 0$
5	$\sum_{i=0}^{n-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$	30/100	$[-30, 30]$	$F(\mathbf{1}) = 0$
6	$\sum_{i=0}^{n-1} \left(\left[x_i + \frac{1}{2} \right] \right)^2$	30/100	$[-100, 100]$	$F(\mathbf{p}) = 0,$ $0.5 \leq p_i < 0.5$
7	$(\sum_{i=0}^{n-1} (i+1)x_i^4) + rand(0, 1)$	30/100	$[-1.28, 1.28]$	$F(\mathbf{0}) = 0$
8	$\sum_{i=0}^{n-1} \left(-x_i \sin \left(\sqrt{ x_i } \right) \right)$	30/100	$[-500, 500]$	$F(\mathbf{420.97}) = -12569.5/-41898.3$
9	$\sum_{i=0}^{n-1} (x_i^2 - 10 \cos(2\pi x_i) + 10)$	30/100	$[-5.12, 5.12]$	$F(\mathbf{0}) = 0$
10	$-20 \exp \left(-0.2 \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} x_i^2} \right) - \exp \left(\frac{1}{n} \sum_{i=0}^{n-1} \cos(2\pi x_i) \right) + 20 + e$	30/100	$[-32, 32]$	$F(\mathbf{0}) = 0$
11	$\frac{1}{4000} \left(\sum_{i=0}^{n-1} x_i^2 \right) + \prod_{i=0}^{n-1} \cos \left(\frac{x_i}{\sqrt{i+1}} \right) + 1$	30/100	$[-600, 600]$	$F(\mathbf{0}) = 0$

(continued)

Table 2.1 (continued)

#	Function	Dimension	Ranges	Minimum
12	$\frac{\pi}{n} \left\{ 10 (\sin (\pi y_1))^2 + \sum_{i=0}^{n-2} \left((y_i - 1)^2 \left(1 + 10 (\sin (\pi y_{i+1}))^2 \right) \right) + (y_n - 1)^2 \right\} + \sum_{i=0}^{n-1} u(x_i, 10, 100, 4),$ <p>where : $y_i = 1 + \frac{1}{4}(x_i + 1)$,</p> $u.(x, a, b, c) = \begin{cases} b(x - a)^c, & \text{if } x > a \\ b(-x - a)^c, & \text{if } x < -a \\ 0, & \text{if } -a \leq x \leq a \end{cases}$	30/100	$[-50, 50]$	$F(-1) = 0$
13	$\sum_{i=0}^{10} \left(a_i - \frac{x_0(b_i^2 + b_i x_1)}{b_i^2 + b_i x_2 + x_3} \right)^2,$ <p>where :</p> $a = (0.1957, 0.1947, 0.1735, 0.1600, 0.0844, 0.0627, 0.0456, 0.0342, 0.0323, 0.0235, 0.0246)$ $b = \left(4, 2, 1, 0.5, 0.25, \frac{1}{6}, \frac{1}{8}, \frac{1}{10}, \frac{1}{12}, \frac{1}{14}, \frac{1}{16} \right)$	4	$[-5, 5]$	$F(0.19, 0.19, 0.12, 0.14) = 0.0003075$
14	$4x_0^2 - 2.1x_0^4 + \frac{1}{3}x_0^6 + x_0x_1 - 4x_1^2 + 4x_1^4$	2	$[-5, 5]$	$F(-0.09, 0.71) = -1.0316$

Table 2.2 Results of evolutionary algorithms on the benchmark problems of dimensionality 30 or less

#	DE		PSO		Standard EA (GA)	
	Mean	Std dev	Mean	Std dev	Mean	Std dev
1	0.00	0.00	0.00	0.00	1.79E-03	2.77E-04
2	0.00	0.00	0.00	0.00	1.72E-02	1.70E-03
3	2.02E-09	8.26E-10	0.00	0.00	1.59E-02	4.25E-3
4	3.85E-08	9.17E-09	2.10E-16	8.01E-16	1.98E-02	2.07E-03
5	0.00	0.00	4.03E+00	4.99E00	3.13E+01	1.74E+01
6	0.00	0.00	4.00E-02	1.98E-01	0.00	0.00
7	4.94E-03	1.13E-03	1.91E-03	1.14E-03	7.11E-04	3.27E-04
8	-1.2569E+04	2.30E-04	-7.19E+0.3	6.72E+0.2	-1.17E+04	2.34E+02
9	0.00	0.00	4.92E+01	1.62E+01	7.18E-01	9.22E-01
10	-1.19E-15	7.03E-16	1.40E+00	7.91E-01	1.05E-02	9.08E-04
11	0.00	0.00	2.35E-02	3.5E-02	4.64E-03	3.96E-03
12	0.00	0.00	3.82	8.40E-01	4.56E-06	8.11E-07
13	4.17E-04	3.01E-04	1.34E-03	3.94E-03	3.70E-04	8.78E-05
14	-1.03E00	1.92E-08	-1.03E00	3.84E-08	-1.03E00	3.16E-08

Table 2.3 Results of evolutionary algorithms on the benchmark problems of dimensionality 100

#	DE		PSO		Standard EA (GA)	
	Mean	Std dev	Mean	Std dev	Mean	Std dev
1	0.00	0.00	0.00	0.00	5.23E-04	5.18E-05
2	0.00	0.00	1.80E+01	6.52E+01	1.74E-02	9.43E-04
3	5.87E-10	1.83E-10	3.67E+03	6.94E+03	3.68E-02	6.06E-03
4	1.13E-09	1.42E-10	5.31E+00	8.63E-01	7.67E-03	5.71E-04
5	0.00	0.00	2.02E+02	7.66E+02	9.25E+01	1.29E+01
6	0.00	0.00	2.10E+00	3.52E+00	0.00	0.00
7	7.66E-03	6.58E-04	2.78E-02	7.31E-02	7.05E-04	9.70E-05
8	-4.1898E+04	1.06E-03	-2.16EE+04	1.73E+03	-3.94E+04	5.36E+02
9	0.00	0.00	2.43E+02	4.03E+01	9.98E-02	3.04E-01
10	8.02E-15	1.74E-15	4.49E+00	1.73E+00	2.93E-03	1.47E-04
11	5.42E-20	0.00	4.17E-01	6.45E-01	1.89E-03	4.42E-03
12	0.00	0.00	1.18E-01	1.75E-01	2.98E-07	2.76E-08

2.2.2 Using Constraints in DEO

The constraints can be used in neural network training where specific requirements are posed to neurons parameters' ranges.

Consider we have an optimization problem with constraints:

Minimize $F(\mathbf{X})$

Subject to:

$$\begin{aligned} G_j(\mathbf{X}) &\leq 0, \quad j = 1, \dots, q, \\ H_j(\mathbf{X}) &= 0, \quad j = q + 1, \dots, m, \\ l_i &\leq x_i \leq u_i, \quad i = 1, \dots, n, \end{aligned}$$

where $\mathbf{X} = (x_1, \dots, x_n)$, $F(\mathbf{X})$ is the objective function, $G_j(\mathbf{X})$ and $H_j(\mathbf{X})$ are the constraint functions, each variable x_i are limited by lower l_i and upper bounds u_i .

Then we can define a constraints violation function $C(\mathbf{X})$ as follows (Takahama and Sakai 2012):

$$C(\mathbf{X}) = \max \left\{ \max_j \{0, G_j(\mathbf{X})\}, \max |H_j(\mathbf{X})| \right\}$$

or

$$C(\mathbf{X}) = \sum_{j=1}^q \left(\max_j \{0, G_j(\mathbf{X})\} \right)^2 + \sum_{j=q+1}^m (H_j(\mathbf{X}))^2$$

Then of the two vectors \mathbf{U} and \mathbf{V} (i.e. potential solutions, elements of a DE population) from the population P , whether \mathbf{U} is better than \mathbf{V} can be decided as follows:

$$\mathbf{U} \succ \mathbf{V} \Leftrightarrow (F(\mathbf{U}) < F(\mathbf{V}) \text{ and } |C(\mathbf{U}) - C(\mathbf{V})| \leq \varepsilon) \text{ or } (C(\mathbf{U}) < C(\mathbf{V}))$$

where $\varepsilon \geq 0$ is a small value.

A found solution \mathbf{X} is considered feasible if $C(\mathbf{X}) \leq \varepsilon$.

As can be seen the goal is to minimize both the constraint violation function and the objective function. Note also that the minimization of the constraint violation function is more important for unfeasible population vectors than the objective function.

2.2.3 Training of All Types of Neural Networks by DEO

As we have already mentioned in previous sections, evolutionary computing based methods are more flexible than classical methods when using for global optimization of functions (Aliiev et al. 2009).

As they do not require any restrictive properties for the functions or the computational models to work with, they can be effectively used for training of parameters of ordinary, fuzzy, and fuzzy type-2 neural networks. The models of neural networks, and especially fuzzy and high-order fuzzy networks can be described by complex nonlinear, non-convex, and non-differentiable functions.

Training of a neural network is in fact a procedure to minimize a function evaluating the network error, e.g. mismatch between the network's actual output and the desired output for a given input. The typical error function is described as follows:

$$E = \frac{1}{n \cdot s_y} \sum_{p=1}^n \sum_{i=1}^{s_y} (y_{pi}^* - y_{pi})^2$$

Here y_{pi}^* is the desired value (target) for output i when we apply input value vector \mathbf{x}_p , y_{pi} is the corresponding output of the model output, n is the number of training patterns, and s_y is the number of outputs in the model.

The decision or optimization variables or parameters for a neural network are either its connection weights (perceptron like NN) or fuzzy parameters describing the input and output linguistic terms in fuzzy rules (NN-based fuzzy inference systems). To apply an evolutionary algorithm, in our case, the DE, the whole bunch of these parameters should have been considered as a population individual.

For example, for a perceptron-like recurrent fuzzy neural network (RFNN), we consider a population individual to represent a whole combination of weights ($\tilde{W} = \{\tilde{w}_{lij}\}$, $\tilde{V} = \{\tilde{v}_{lij}\}$) and biases ($\tilde{\theta} = \{\tilde{\theta}_{li}\}$) (i.e. parameters of RFNN) defining the input/output mapping (Aliev et al. 2009). For a type-2 neural network, considered in Sect. 3.6.2 of this book, each fuzzy term-parameter is itself described by several sub-parameters: the parameters LL , LR , ML , MR , RL , RR for all input terms and the parameters L , ML , MR , R for all outputs terms (Aliev et al. 2011).

The population maintains a number of potential parameter sets defining different network solutions and recognizes one of these solutions to be the best solution. This best solution is the one with minimum training error. After a series of generations, the best solution may converge to a near-optimum solution, which would represent a network performing with the required accuracy.

The detailed DE based algorithm for training of FNN and FRNN has been presented in Sect. 3.5.2 of this book. Section 3.9.1 presents a DE based training for a type-2 neural network, considered in Sect. 3.6.2 of this book.

References

- Aliev RA, Guirimov BG, Fazlollahi B, Aliev RR (2009) Evolutionary algorithm-based learning of fuzzy neural networks. Part 2: Recurrent fuzzy neural networks. Fuzzy Sets and Systems archive, Volume 160 Issue 17, 2553–2566.
- Aliev RA, Pedrycz W, Guirimov B, Aliev RR, Ilhan U, Babagil M, Mammadli S (2011) Type-2 fuzzy neural networks with fuzzy clustering and differential evolution optimization. Information Sciences, Volume 181 Issue 9, 1591–1608.
- Bonabeau E, Dorigo M, Theraulaz G (1999) Swarm intelligence: from natural to artificial systems. Oxford University Press, USA

- Brest J, Greiner S, Boskovic B, Mernik M, Zumer V (2006) Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation*, Vol. 10, No 6.
- Chakraborty UK (eds) (2008), *Advances in Differential Evolution*, Springer
- Chiong R, Weise T, Michalewicz Z (eds) (2012) *Variants of evolutionary algorithms for real-world applications*. Springer
- Clerc M (2006) *Particle swarm optimization*. ISTE
- Crosby JL (1973) *Computer simulation in genetics*. John Wiley & Sons, London
- Dorigo M, Stützle T (2004) *Ant colony optimization*. MIT Press
- Eiben A, Smith J (2003). *Introduction to evolutionary computing*. Springer
- Feoktistov V (2006) *Differential evolution: in search of solutions*. Springer
- Geem ZW, Kim JH, Loganathan GV (2001) A new heuristic optimization algorithm: harmony search. *Simulation* 76: 60–68
- Goldberg DE (1989) *Genetic algorithms in search, optimization and machine learning*. Addison Wesley
- Griewangk (1981), A.O., Generalized Descent for Global Optimization, *JOTA*, vol. 34, pp. 11–39.
- Karahan H, Gurarslan G, Geem ZW (2012) Parameter estimation of the nonlinear Muskingum flood routing model using a hybrid harmony search algorithm. *Journal of Hydrologic Engineering*. doi:[10.1061/\(ASCE\)HE.1943-5584.0000608](https://doi.org/10.1061/(ASCE)HE.1943-5584.0000608)
- Kennedy J, Eberhart R (1995) Particle swarm optimization. In: *Proceedings of IEEE International Conference on Neural Networks IV: 1942–1948*. doi:[10.1109/ICNN.1995.488968](https://doi.org/10.1109/ICNN.1995.488968)
- Kennedy J, Eberhart R (2001) *Swarm intelligence*. Morgan Kaufmann Publishers, San Francisco
- Langdon WB, Poli R (2002) *Foundations of genetic programming*. Springer-Verlag
- Price K, Storn, RM, Lampinen JA (2005). *Differential evolution: a practical approach to global optimization*. Springer
- Reynolds RG (1994) An introduction to cultural algorithms. In: *Proceedings of the 3rd Annual Conference on Evolutionary Programming*. World Scientific Publishing: 131–139
- Ricart J, Hüttemann G, Lima J, Barán B (2011) Multiobjective harmony search algorithm proposals. *Electronic Notes in Theoretical Computer Science*
- Storn R, Price K (1997) Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization* 11: 341–359. doi:[10.1023/A:1008202821328](https://doi.org/10.1023/A:1008202821328)
- Takahama T, Sakai S (2012) Efficient Constrained Optimization by the ϵ Constrained Rank-Based Differential Evolution. In: *Proc. of 2012 I.E. Congress on Evolutionary Computation (CEC)*.
- Vesterstrøm J, Thomsen R (2004) A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical Benchmark Problems. In: *Congress on Evolutionary Computation (CEC2004)*, Volume:2, 1980–1987.
- Yao X, Liu Y (1996) Fast evolutionary programming. In: Fogel LJ, Angeline PJ, Back T (eds), *Proceedings of the 5th Annual Conference on Evolutionary Programming*, 451–460. MIT Press.

Type-2 Fuzzy Neural Networks and Their Applications

Aliev, R.A.; Guirimov, B.G.

2014, XIII, 190 p. 73 illus., Hardcover

ISBN: 978-3-319-09071-9