Incorporation Family Competition into Gaussian and Cauchy Mutations to Training Neural Networks Using an Evolutionary Algorithm

Jinn-Moon Yang
Department of Computer Science
and Information Engineering,
National Taiwan University,
Taipei, Taiwan
moon@csie.ntu.edu.tw

Jorng-Tzong Horng
Department of Computer Science
and Information Engineering,
National Central University,
ChungLi, Taiwan
horng@db.csie.ntu.edu.tw

Cheng-Yen Kao
Department of Computer Science
and Information Engineering,
National Taiwan University,
Taipei, Taiwan
cykao@csie.ntu.edu.tw

Abstract- This paper presents an evolutionary technique to train neural networks in tasks requiring learning behavior. Based on family competition principles and adaptive rules, the proposed approach integrates decreasing-based mutations and self-adaptive mutations. Different mutations act global and local strategies separately to balance the trade-off between solution quality and convergence speed. The algorithm proposed herein is applied to two different task domains: Boolean functions and artificial ant problem. Experimental results indicate that, in all tested problems, the proposed algorithm performs better than other canonical evolutionary algorithms, such as genetic algorithms, evolution strategies, and evolutionary programming. Moreover, essential components such as mutation operators and adaptive rules in the proposed algorithm are thoroughly analyzed.

1 Introduction

As widely recognized, artificial neural networks (ANNs)[9] achieve complex computational tasks, such as language recognizer, autonomous robotic control [13], and time serial prediction [10]. In addition to having the approximation capabilities for multilayer feedforward networks in numerous functions [7]. ANNs avoid the bias of a designer in shaping system development owing to their flexibility, robustness, and tolerance of noise. To train ANNs is usually formulated as a weight training process. The process is performed to achieve an optimal set of connection weights for a network according to some optimal criteria. Back propagation [12], a conventional training algorithm, implements a gradient decent search algorithm, which attempts to minimize the total error between actual output and target output of an ANN. However, back propagation is susceptible to being trapped into local optima and is inefficient in terms of searching for a global minimum of a function which is vast, multimodal, and non-differentiable.

As global search approaches, evolutionary algorithms effectively deal with complex and nondifferentiable search space. Pertinent research [11], [19] has demonstrated that the search speed of evolutionary algorithms is comparable to back propagation if genetic operators are well designed. Evolutionary algorithms train or evolve various ANNs structures for many application domains.

Evolutionary methodologies can be categorized as genetic algorithms [6], evolutionary programming, and evolution strategies. Applying genetic algorithms to train neural networks may be unsatisfactory because recombination operators incur several problems such as competing conventions [15] and the epistasis effect [2]. Epistasis, a nonlinear interaction, dramatically retards genetic algorithms. To ensure a better performance, modified approaches, called real-coded genetic algorithms, use real-valued representation and promote the ability of mutation operators to reduce the above drawbacks. However, these real-coded genetic algorithms employed random mutations so that they make a larger jump in a search space; however, this may be insufficient to achieve good solution quality. On the other hand, evolution strategy and evolutionary programming use real-valued representation and focus on self-adaptive Gaussian mutation. Despite successful implementation of the mutation operator for various numerical optimization problems and its reputation as a good operator for local search, self-adaptive Gaussian mutation does not perform well for certain specific functions and it is easily trapped to local optima for rugged functions [21], [20].

This paper presents an evolutionary algorithm, called Family Competition Evolutionary Algorithm, hereafter called FCEA, to train neural networks. The proposed algorithm combines four mutation operators: self-adaptive Gaussian mutation, self-adaptive Cauchy mutation, decreasing-based Gaussian mutation, and decreasing-based Cauchy mutation. FCEA constructs a relationship among these four operators to balance the search power of the exploration and exploitation by applying family competition and by automatically controlling the step sizes of mutations. These operators compensate for their disadvantages to enhance the performance of FCEA. To our knowledge, FCEA is the first approach to successfully integrate self-adaptive mutations with decreasing-based mutations via our efficient adaptive rules based on family competition principles.

The proposed algorithm is applied to two different problem areas: Boolean functions learning[12] and an artificial ant problem [8], [14]. First, FCEA is applied to solve two famous Boolean function problems, i.e., Xor and 2-bit adder, in order to compare with previous results. Then, the algo-

0-7803-5536-9/99/$10.00 ©1999 IEEE

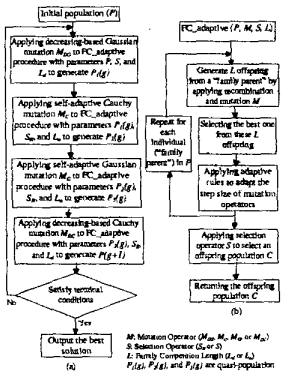


Figure 1: Overview of our algorithm: (a) FCEA (b) FC_adaptive procedure.

rithm proposed herein trains networks to learn how to generate different tracks based on sensory inputs of an ant robot. Our FCEA algorithm performs better than genetic algorithms, evolution strategies, and evolutionary programming in all two problems. This work also thoroughly analyzes the essential components of FCEA such as mutation operators and step sizes. Also investigated herein is the influence of the adaptive rules and strategy parameters of the proposed algorithm.

2 Family Competition Evolutionary Algorithm

Proposed herein Family Competition Evolutionary Algorithm is a multi-operator approach. FCEA incorporates four mutation operators: decreasing-based Gaussian mutation, self-adaptive Gaussian mutation, self-adaptive Cauchy mutation, and decreasing-based Cauchy mutation. Fig.1 depicts the flow of FCEA. Each block in Fig.1 indicates the use of a mutation operator and FC_adaptive shown in Fig.1(b) to generate a population of offspring. A factor that determines the perturbation size significantly affect the power of these four mutation operators. This important factor is called step size. These four mutation operators are the main operators of FCEA; they are sequentially applied in four stages.

The FCEA in Fig. 1 work as follows. Initially, N networks are generated. The fitness value of each network is evaluated. FCEA then enters the main evolutionary loop consisting of four stages: decreasing-based Gaussian mutation stage, self-adaptive Cauchy mutation stage, self-adaptive Gaussian mutation stage, and decreasing-based Cauchy mutation stage. Each stage is realized by calling FC_adaptive procedure illustrated in Fig. Fig. 1(b).

The FC_adaptive procedure uses four parameters, i.e., parent population (P), mutation operator (M), selection operator (S), and family competition length (L), to generate a new quasi-population which becomes the parent population of the next stage. The kernel of FC_adaptive consists of family competition and adaptive rules for step sizes. In the family competition, each individual in the population sequentially becomes a "family father". Herein, the term "family father" is used to distinguish other terms such as parent, because a family is built on the basis of the "family father" in the family competition procedure. Next, the "family father" and other individual selected from the population are applied by the recombination operator and mutation operator M to generate an offspring. The process is repeated according to the family length L. A family with L offspring via the"family father" is then built. These L offspring in each family then compete with each other and the one with the best objective value survives. These adaptive rules are applied to adapt the step-size vector of this individual for mutation operators. Therefore, the size of each new quasi-population remains N. Finally, the selection operator (S) chooses the N fittest individuals from the set of parent population and they become the parent population of the next stage. The following subsections describe the components of the FCEA approach including the chromosome representation, family competition, recombination operators and mutation operators, selection methods, and control rules.

2.1 Chromosome representation and initialization

Each network is represented as a quadruple n-dimensional vector $(\vec{x}, \vec{\sigma}, \vec{v}, \vec{\psi})$, where n denotes the number of connection links of an ANN. The vector $\vec{x}$ is an optimized variable vector, i.e., a weight vector of the connection links of an ANN. In addition, $\vec{\sigma}$, $\vec{v}$, and $\vec{\psi}$ represent the step-size vectors of decreasing-based mutations, self-adaptive Gaussian mutation, and self-adaptive Cauchy mutation, respectively. Herein, the initial value of each entry of $\vec{x}$ is randomly chosen over $[-0.1, 0.1]$ and the initial values of each entries of the vectors $\vec{\sigma}$, $\vec{v}$, and $\vec{\psi}$, are set to be 1.0, 0.25, and 0.25, respectively. In the upcoming subections, we use $\vec{a} = (\vec{x}_a, \vec{\sigma}_a, \vec{v}_a, \vec{\psi}_a)$ to represent an individual called "family father" and $\vec{b} = (\vec{x}_b, \vec{\sigma}_b, \vec{v}_b, \vec{\psi}_b)$ to denote another parent. The offspring $\vec{c} = (\vec{x}_c, \vec{\sigma}_c, \vec{v}_c, \vec{\psi}_c)$ is a generated offspring by applying the recombination or mutation operators. The symbol $x_j^{\vec{d}}$ denotes the j-th connection weight of the individual $\vec{d}$.

2.2 Family Competition

The family competition in FCEA can be viewed as a local search procedure and works as follows. An individual, referred to as "family parent", is the leading role of genetic operators. The "family parent" generates offspring by using recombination operators with probability p_c and mutation operator with probability 1. While the recombination is applied, recombination selection is used to select two parents: one is the "family parent" and other individual randomly selected from population. Recombination generates only one offspring $\vec{c}$. The offspring $\vec{c}$ is exact same the "family parent" if recombination operators are not applied. Then, mutation operator(M) is applied to the offspring $\vec{c}$ to generated an offspring $\vec{c}'$. The "family parent" generates L offspring

by repeatedly applies these procedures. These L offspring compete with each other and only the one with best fitness survives. FCEA employs this strategy to avoid premature convergence by maintaining the diversity of the populations because the L offspring generated from the same "family parent" may resemble each other. Family competition principle is that each individual in the population sequentially becomes the "family father" and perform the local search to generate L offspring; and then only the one with best fitness survives. Therefore, FCEA will generate $L \cdot N$ offspring in each stage so that FCEA generates $2 \cdot (L_d + L_a)$ offspring in one generation.

2.3 Recombination Operators

FCEA uses three kinds of recombination operators: modified discrete recombination, blend crossover (BLX-0.5) [3] and intermediate recombination [1]. The intermediate recombination is a special case of BLX-0.5.

Modified discrete recombination: The original discrete recombination [1] generates a child that inherits genes from two parents with equal probability. Herein, this recombination is modified such that a child inherits genes from the "family father" $\vec{a}$ with probability 0.8 and from another parent $\vec{b}$ with probability 0.2. The modified discrete recombination is given below.

$$x_j^c = \begin{cases} x_j^a \text{ with probability } 0.8 \\ x_j^b \text{ with probability } 0.2. \end{cases} \quad (1)$$

The probabilities in (1) can reduce the undesired effects of competing conventions on training neural networks.

BLX-0.5 and intermediate recombination: The BLX-0.5 [3] is successfully used in a real-coded genetic algorithm. It is defined as follows:

$$\varpi_j^c = \varpi_j^a + \beta(\varpi_j^b - \varpi_j^a), \quad (2)$$

where ϖ may be any vector such as $\vec{x}$, $\vec{\sigma}$, $\vec{v}$,or $\vec{\psi}$ and β is chosen uniformly from the range [-0.5, 1.5]. BLX-0.5 is called intermediate recombination when β is equal to 0.5. This is accounts for why intermediate recombination is considered herein to be a special case of BLX-0.5.

This work follows the work of the evolution strategies community to employ only intermediate recombination on step-size vectors, i.e., $\vec{\sigma}$, $\vec{v}$, and $\vec{\psi}$. Meanwhile, FCEA applies discrete recombination, BLX-0.5, and intermediate recombination to recombine connection links $\vec{x}$. In the following experiments, the probabilities are 0.2, 0.1 and 0.1, respectively.

2.4 Mutation Operators

Mutations are main operators of our FCEA. As mentioned earlier, four mutation operators are used in FCEA. Details of each operator are described as follows.

Self-adaptive Gaussian mutation: Schwefel [17] proposed a self-adaptive technique, called self-adaptive Gaussian mutation. This technique performs well in parameter optimization problems. It is accomplished by first mutating step size v_j. Next the connection link x_j is mutated by adding a normally distributed random value with zero and v_j as expectation and standard deviation, respectively. This operator is realized by using the following equations (3) and (4).

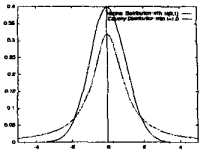


Figure 2: Comparisons of Gaussian and Cauchy distributions.

$$v_j^c = v_j^a \cdot [\tau' \cdot N(0,1) + \tau \cdot N_j(0,1)] \quad (3)$$
$$x_j^c = x_j^a + v_j^c \cdot N_j(0,1), \quad (4)$$

where $N(0,1)$ is a normal distribution with mean 0 and standard deviation 1. The solid line in Fig. 2 shows the density distribution of $N(0,1)$. In addition, $N_j(0,1)$ is a normalization distribution for the jth connection link. In our experiments, τ and τ' are set to $(\sqrt{2n})^{-1}$ and $\sqrt{2\sqrt{n}})^{-1}$, respectively.

Self-adaptive Cauchy mutation: Cauchy density distribution is the dash line in Fig. 2 and is defined as follows:

$$f_t(x) = \frac{t/\pi}{t^2 + x^2}, \quad -\infty \prec x \prec \infty, \quad (5)$$

where t is a scale parameter [21]. The behavior of self-adaptive Cauchy mutation is exactly the same as self-adaptive Gaussian mutation except Cauchy distribution replaces the normal distribution. Restated, the step size is controlled by using the similar equation in (6) and then the connection link x_j is mutated by adding a Cauchy distributed random value with ψ_j as standard deviation. Self-adaptive Cauchy mutation is given by (6) and (7).

$$\psi_j^c = \psi_j^a \cdot exp[\tau' \cdot N(0,1) + \tau \cdot N_j(0,1)] \quad (6)$$
$$x_j^c = x_j^a + \psi_j^c \cdot C_j(t), \quad (7)$$

where $C(t)$ is a Cauchy probability distribution function with parameter t. In our experiments, t is 1.

Decreasing-based mutations: The decreasing-based Gaussian mutation and decreasing-based Cauchy mutation share the same step-size vector $\vec{\sigma}$. It is decreased by a decreasing rate γ, $0 \prec \gamma \prec 1$. These two mutations use two following (9) and (10) to mutate connection links, respectively.

$$\sigma_j^c = \gamma \cdot \sigma_j^a \quad (8)$$
$$x_j^c = x_j^a + \sigma_j^c \cdot N_j(0,1) \quad (9)$$
$$x_j^c = x_j^a + \sigma_j^c \cdot C_j(1), \quad (10)$$

where γ is 0.95 in our experiments. According to (3), (6), and (8), two interesting phenomena are observed. First, (8) can

save computational time because it is multiplication; in addition, (3) and (6) must compute a normal distribution function as well as an exponential function. Second, the search behavior of decreasing-based mutation markedly differs from self-adaptive mutations because (8) decreases the step sizes by a fixed rate; however (3) and (6) adapt step sizes by a stochastic approach.

2.5 Selections

FCEA uses four selections: recombination selection, family selection, replacement selection, and population selection. FCEA employs recombination selection to select two individuals for recombination. One is "family father" and the other is randomly selected from the population. Family selection selects the one with best objective value from the L offspring that are generated from the same "family father". The best children population set is then formed by repeatedly applying the procedure. In our three mutation stages except for the decreasing-based Gaussian mutation stage, FCEA employs replacement selection to select the one with better objective value from "family father" and its best child that is selected by family selection. Combining family selection and replacement selection is usually viewed as a local search procedure. Population selection selects the best N individuals from the union set formed by the parent population set and best children population set. Population selection resembles $(\mu + \mu)$-ES used by traditional evolution strategies.

2.6 Adaptive Rules

Controlling the step size heavily influences the performance of Gaussian and Cauchy mutations. FCEA constructs the relationship between self-adaptive mutations and decreasing-based mutations by combining deterministic, self-adaptive, and adaptive techniques to effectively control the step sizes of Gaussian and Cauchy mutations according to the adaptation classification [5]. Herein, these rules are summarized into A-rules, including A-adaptive-rule and A-decrease-rule, for self-adaptive mutations and D-rules, including D-decrease-rule and D-increase-rule, for decreasing-based mutations.

1. **A-rules:**
   - **A-adaptive-rule:** This self-adaptive rule controls the step sizes of $\vec{v}$ and $\vec{\psi}$ according to (3) and (6). It is called a self-adaptive rule because the step-size vectors $\vec{v}$ and $\vec{\psi}$ are directly encoded into a chromosome of an individual and undergo mutations and recombination. The rule is applied when the mutation is a self-adaptive one.
   - **A-decrease-rule:** The rule decreases the step-size vectors $\vec{v}$ and $\vec{\psi}$ of a "family parent" when the "family parent" is better than its best child generated by applied family competition. Step sizes $\vec{v}$ and $\vec{\psi}$ are adapted while self-adaptive Gaussian and self-adaptive Cauchy mutation are applied, respectively. The step sizes $\vec{v}$ and $\vec{\psi}$ are adapted in the following manner:

$$\varpi_j^a = \gamma\varpi_j^a \text{ if "family parent" } \vec{a} \text{ is better than its best child,} \quad (11)$$

     where γ is the decreasing rate and γ is 0.95 in our experiments.

2. **D-rules:**
   - **D-decrease-rule:** The rule is a deterministic rule because it decreases the step size $\vec{\sigma}$ according to (8). The rule is applied when the mutation is a decreasing-based one.
   - **D-increase-rule:** This adaptive rule enlarges the step size $\vec{\sigma}$ of the best child when family competition is applied and the best child is better than its "family father" in two self-adaptive mutation stages. It updates the step sizes as follows:

$$\sigma_j^c = \beta v_{mean}^c \; if \; \sigma_j^c \prec \beta v_{mean}^c \text{ and the best child } \vec{c} \text{ is better than its "family parent" } \vec{a}, \quad (12)$$

     where $\vec{v}$ is the step-size vector of the best child; v_{mean}^c is the mean value of the vector $\vec{v}$; and β is 0.2 in our experiments.

FCEA successfully combines self-adaptive mutations and decreasing-based mutations via A-rules and D-rules to enhance the performance. Later we demonstrate how these rules can enhance the performance of FCEA.

3 Boolean Functions Learning

FCEA is applied to optimize the connection weights for two well-known Boolean function problems [12] . To compare with previous works, FCEA uses standard fully connected networks structures which have a hidden layer with a bias neuron. These two problems are described as follows:

1. **Xor:** An ANN has 2 input nodes, 2 hidden nodes, and 1 output node. There are 9 connection weights and 4 input patterns. The output value is the Exclusive OR of the input bits.
2. **Addition:** An ANN has 4 input nodes, 4 hidden nodes, and 3 output nodes. These are 35 connection weights and 16 input patterns. The output pattern is the result of the sum of the two 2-bits input strings.

Herein, binary input patterns are used and a network is trained to generate output values ranging from 0 to 1. The fitness function of a network is based on mean square error and is given below

$$\frac{1}{mN_0}\sum_{k=1}^{m}\sum_{j=1}^{N_0}(O_{kj} - O'_{kj})^2 \quad (13)$$

Table 1: Comparison the results of FCEA with previous works on two Boolean functions.

Method	Xor	Addition
FCEA	2206 100%	256464 96%
Evolutionary Programming [4]	2000.0 (100%)	N/A
Standard Genetic Algorithm [18]	6120 (80%)	N/A
Adaptive Genetic Algorithm [18]	3673 (100%)	N/A
GENITOR [19]	500 (100%)	2000000 N/A (56%)
GENITOR II [19]		2000000 (93%)

†GENITOR is a well-known modified genetic algorithm.

†GENITOR II is a distributed version of GENITOR.

†(N/A denotes not available in the literature.)

†The values in () is the successful classified rate.

where O_{kj} and O'_{kj} denote, respetively, the output value and training value of the jth output neuron for the kth input pattern; m is the number of input pattern; and N_o is the number of output neuron. A training input pattern is classified correctly if the tolerance of $|O_{kj} - O'_{kj}|$ is below 0.1 for each output neuron. A network is convergent if the network classifies all the training input patterns.

Evolution begins by initializing all the connection weights $\vec{x}$ of each network to random values between -0.1 and 0.1. The initial values of step sizes for decreasing-based mutations, self-adaptive Gaussian mutation, and self-adaptive Cauchy mutation are 1.0, 0.25, and 0.25, respectively. The family competition length L_d and L_a in the decreasing-based stages and self-adaptive stages are 3 and 9, respectively. In this case, FCEA generates 720 networks, i.e. (3+9+9+3)·30, in one generation if the population size is 30. The population size is 10 for Xor and is 30 for addition problems. The rate of recombination is 0.2. These parameter values except for the population size are applied to all problems addressed herein.

Table 1 compares our FCEA, evolutionary programming [4], and genetic algorithm [18], [19] on the Boolean functions. Detailed implementation of these compared approaches can be found in the original papers. According to pertinent literature, the performance of their evolutionary algorithms is competitive with back propagation. FCEA is executed 50 runs for each problem and is up to 500000 function evaluations, i.e., the number of generated offspring, for each run. FCEA can solve all Boolean functions within reasonable function evaluations; the successful classified rates are 96% for Addition problem.

Standard evolutionary algorithms, such as simple genetic algorithm [18] and (1+6)-ES [16], cannot completely solve Xor problem for all runs. The modified evolutionary algorithms [16], [18] can resolve simple problems, such as Xor.

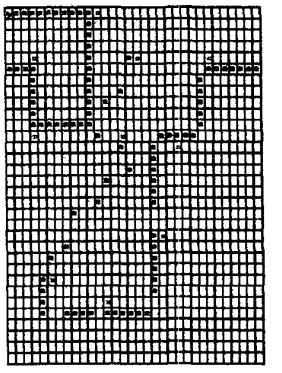


Figure 3: Artificial ant problems: "John Muir Trail".

However, they only solve several simple problems. GENITOR needed only around 500 recombination to resolve Xor problem. However, it required a population of 5000 and 2 million function evaluations to solve 2-bit adder and the classified rate is only 56%. These results indicates that although efficient for simple problems, these evolutionary algorithms can not solve complicated problems, such as Addition problems. GENITOR II, a distributed version of GENITOR, can increase classified rate to 93% in the Addition problem. However, its population size is also 5000 and the number of function evaluations also reaches 2 million. In contrast to these approaches, FCEA only needs 256464 function evaluations and the successfully classified rate is up to 96% by using small population size, i.e., 30, for Addition problem. These results demonstrate that FCEA is a robust approach to train forward networks for Boolean functions learning.

4 The Ant Problem

This study applies FCEA to experiment on complex search and collection task that is the tracker task "John Muir Trail" [8]. In this problems, a simulated ant is placed on a two-dimensional toroidal grid that contains a trail of food. The ant traverses the grid to collect any food encountered along the trail. This task attempts to train a neural network, i.e., a simulated ant, that collects the maximum number of pieces of food during the given time steps. Fig.3 shows this trail. Each black box in the trail stands for a food unit. According to the environment of [8], the ant stands on one cell, facing one of the cardinal directions; it can sense only the cell ahead of it. After sensing the cell ahead of it, the ant must take one of four actions: move forward one step, turn right 90°, turn left 90°, and no-op (do nothing). In the optimal trail of the "John Muir Trail", there are 89 food cells, 38 no food cells, and 20 turns. So, the number of minimum steps for eating all food is 147 time steps. On the other hand, an ant requires at least 165 time steps to completely travel the optimal trail of the "Santa Fe Trial".

To compare with previous research, we follow the work

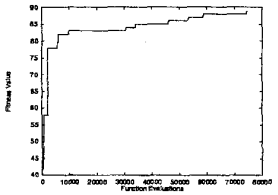


Figure 4: The typical convergent curve of "John Muir Trail" problems.

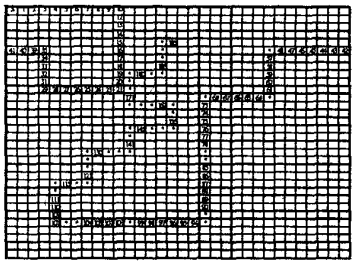


Figure 5: The typical search behavior of a simulated ant controlled by our evolved neural controller for "John Muir Trail" ant problem.

of [8]. That investigation not only used finite state machines and recurrent neural networks to represent the problem, but also used the traditional bit-string genetic algorithm to train the structures. Each simulated ant is controlled by a network having two input nodes and four output nodes. The "food" input is 1 when food is present in the cell ahead of the ant; and the second "no-food" is 1 in the absence of food in the cell in front of the ant. Each output unit corresponds to a unique action: move forward one step, turn right 90°, turn left 90°, or no-op. Each input node is connected to each of the five hidden nodes and to each of the four output nodes. The five hidden nodes are fully connected in the hidden layer. Therefore, this structure is a full connection with shortcut recurrent neural network; its total number of links with bias input is 72. To compare with previous results, the fitness is defined as the number of pieces of food eaten within 200 time steps for "John Muir Trail".

Fig.4 displays the convergence curve of the ant problems. Fig.4 indicates that FCEA only requires about 12,000 function evaluations to train a neural controller to find 82 food pieces within 200 time steps. To find 85 and 88 food pieces within 200 time steps, FCEA then requires about 35000 and 58000 function evaluations. FCEA on average found 81, 87, and 88 food pieces within 200 time steps about 20000, 65000, and 80000 function evaluations, respectively. "John Muir Trail" was tested over 25 runs and the rate of success of finding 89 food pieces was 80%. The remaining 20% of runs the ant foraged at least 86 food pieces. The successful rate can be improved to 96% when the population is 100 and the number of function evaluations is 500,000.

Table 2: Comparison among genetic algorithm, evolutionary programming, and our FCEA on "John Muir Trail" ant problem.

method	population size	average FE	best Performance	Average Performance
Genetic Algorithms[8]	65536	6,553,600	89	N/A
Evolutionary programming[14]	100	184,250	83	82
FCEA	50	126,000	89 (20/25)	88.68
	100	284,000	89 (24/25)	88.96

Fig.5 depicts a typical search behavior and the traveled path of a simulated ant that is controlled by our evolved neural network. The number of the cell is the time step to eat the food. The symbol '*' denotes a cell traveled by an ant when the cell is empty. Fig.5 indicate that the ant requires 195 time steps to seek all 89 food pieces in the environment of "John Muir Trail".

Table 2 compares our FCEA, evolutionary programming [14], and genetic algorithm [8] on the "John Muir Trail" ant problem. Jefferson et al. used traditional genetic algorithms to solve "John Muir Trail". That investigation encoded the problem with 448 bits and used a population of 65536 to achieve the task in 100 generations. Their approach required 6,553,600 networks to forage 89 food pieces exactly within 200 time steps. In contrast to Jefferson's solution, our FCEA uses population sizes 50 and 100, and only requires about 126,000 and 284,000 function evaluations, respectively, to eat 89 food pieces within 195 time steps. Table 2 also indicates that FCEA performs better than evolutionary programming.

5 The Characteristics of FCEA

In this section, we briefly discussed several characteristics of FCEA via experimental designs. Table 3 compares the ten approaches in term of 2-bits Adder functions and an ant problem. Each approach is a combination of operators applied in our FCEA: decreasing-based Gaussian mutation (M_{DG}), self-adaptive Cauchy mutation (M_C), self-adaptive Gaussian mutation (M_G), and decreasing-based Cauchy mutation (M_{DC}). For example, the M_C approach only uses self-adaptive Cauchy mutation; the $M_{DG} + M_C$ approach integrates decreasing-based Gaussian mutation with self-adaptive Cauchy mutation and it also applied the control rules. The FCL1_{FCEA} approach is unique case of our FCEA because the family competition lengths (L_d and L_a) is set to 1. The NCR_{FCEA} approach is also a unique case of our FCEA but it does not apply adaptive rules, i.e., A-decrease-rule and D-increase-rule. The final approach in Table 3 is a standard evolution strategy i.e., $(\mu + \lambda)$-ES, where μ is 20 and λ is 120. Each approach executes 50 runs for Boolean Functions; and 25 runs for the ant problem. The maximum numbers of function evaluations of each run on Boolean functions and the ant

Table 3: Various approaches of operators of FCEA are compared

Methods	Addition	Jefferson's Ant Problem
FCEA FE	256464	156087
SR	96%	20/25 (88.68)
M_C FE	500000	196309
SR	0%	8/25 (87.44)
M_G FE	500000	227511
SR	0%	3/25 (85.50)
M_{DG} FE	483891	242000
SR	3%	1/25 (84.72)
$M_{DG} + M_C$ FE	337987	202483
SR	56%	9/25 (87.92)
$M_{DG} + M_G$ FE	364897	215566
SR	68%	6/25 (87.2)
$FCL1_{FCEA}$ FE	500000	241929
SR	0%	3/25 (85.16)
NCR_{FCEA} FE	379736	204770
SR	66%	12/25(88.12)
$(\mu + \lambda)$-ES FE	345454	222826
SR	46%	6/25 (87.48)

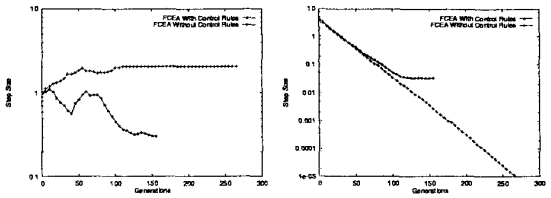
†SR denotes that the percent of an approach finds global optima.

†FE denotes the number of function evaluations

problem are 500,000 and 250,000, respectively. The value in the parenthesis in the ant problem denotes the average number of food pieces eaten.

We observe several properties according to these experimental results of Table 3 and Fig. 6.

- Each mutation operator in FCEA has different performance on the selected problems. These results indicate that each operator has different search behavior.
- Generally, the approaches of a combination of multiple mutations perform better than the approaches of unary-operator mutation and they do not increase proportionally on the number of function evaluations. For example, our FCEA that combines M_{DG}, M_C, M_G, and M_{DC} has the best performance among all approaches on all testing problems. Nevertheless, the number of function evaluations of FCEA is not larger than other approaches for all testing problems.
- The control rules of step sizes are useful because NCR_{FCEA} performs worst than FCEA. Fig. 6(b) indicates that the step size (σ) of decreasing-based mutation becomes small while FCEA does not apply D-increase-rule. Fig. 6(a) indicates that the step size of self-adaptive Gaussian mutation is too large to improve solution while FCEA does not apply A-decrease-rule.
- The family competition length is a one of critical factors of FCEA to obtain better performance for complex problems. For example, FCEA have to enlarge the length in order to solve ant problems.
- Cauchy mutations perform better than Gaussian mutations on training neural networks.



(a) Self-adaptive mutation (b) Decreasing-based mutation

Figure 6: The comparison of average step size between FCEA with adaptive rules and FCEA without adaptive rules on ant problem

6 Conclusions

This study has demonstrated that FCEA is an efficient approach for training neural networks. The proposed algorithm combines decreasing-based mutations with self-adaptive mutations to enhance the performance based on family competition and adaptive rules. Our FCEA is able to balance the exploitation and exploration of search ability. Results from Boolean functions and an ant problems confirm the flexibility and robustness of such an evolutionary approach.

A global optimization method must consist of both global and local search strategies. For our FCEA, the decreasing-based mutation with large initial step size are global search strategies and self-adaptive mutations with family competition procedure and replacement selection are local search strategies. Cauchy mutations are attention to be used in global search strategies than Gaussian mutations as demonstrated in the proposed approach. These mutation operators can be integrated to closely cooperate with each other. These smoothly integrated strategies make our FCEA applicable to train neural networks for various applications as well as to solve various numeric optimization problems. Under appropriate conditions, FCEA is able to converge to a global solution.

In summary, experiments in these well-known problems verify that the proposed approach consistently performs more robustly than other algorithms, such as genetic algorithms, evolution strategies, and evolutionary programming. We believe that the flexibility and robustness of our FCEA makes it a highly effective global optimization tool.

Bibliography

[1] T. Bäck, F. Hoffmeister, and H-P. Schwefel. A survey of evolution strategies. In *Proc. Fourth Int. Conf. on Genetic Algorithms*, pages 2–9, 1991.

[2] Y. Davidor. Epistasis variance: Suitability of a representation to genetic algorithms. *Complex Systems*, 4:368–383, 1990.

[3] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval-schemata. In L. D. Whitley, editor, *Foundations of Genetic Algorithm*, volume 2, pages 187–202. Morgan Kaufmann Publishers, Inc., 1993.

[4] D. B. Fogel, L. J. Fogel, and V. W. Porto. Evolving neural networks. *Biological Cybernetics*, 63:487–193, 1990.

[5] R. Hinterding, Z. Michalewicz, and A. E. Eiben. Adaptation in evolutionary computation: A survey. In *Proc. of IEEE Conf. on Evolutionary Computation*, pages 65–69, 1997.

[6] John H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[7] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257, 1991.

[8] D. Jefferson, R. Collins, C. Cooperand M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The genesys/tracker system. In *Artificial Life II: Proc. of the Workshop on Artificial Life*, pages 549–577, 1990.

[9] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, 1987.

[10] J. R. McDonnell and D. Waagen. Evolving recurrent perceptrons for time-series modeling. *IEEE Trans. on Neural Networks*, 5(1):24–38, 1994.

[11] D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proc. of Eleventh Int. Joint Conf. on Artificial Intelligence*, pages 762–767, 1989.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, pages 318–362. Cambridge, MA: MIT Press, 1986.

[13] R. Salomon. Scaling behavior of the evolution strategy when evolving neuronal control architectures for autonomous agents. In P. J. Angeline et al., editor, *the Lecture Notes in Computer Science: Evolutionary Programming VI*, pages 47–57, 1997.

[14] P. J. Angeline G. M. Saunders and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Trans. on Neural Networks*, 5(1):54–65, 1994.

[15] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proc. of Int. workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37, 1992.

[16] M. Scholz. A learning strategy for neural networks based on a modified evolutionary strategy. In *Parallel Problem Solving from Nature-Proc. 1st Workshop PPSN I (Lecture Notes in Computer Science)*, volume 496, pages 314–318, 1991.

[17] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. Chichester: Wiley, 1981.

[18] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Trans. Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[19] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14:347–361, 1990.

[20] J. M. Yang, C. Y. Kao, and J. T. Horng. A continuous genetic algorithm for global optimization. In *Proc. of the Seventh Int. Conf. on Genetic Algorithms*, pages 230–237, 1997.

[21] X. Yao and Y. Liu. Fast evolutionary programming. In *Proc. of the Fifth Annual Conf. on Evolutionary Programming*, pages 451–460, 1996.