A Tensor-based Mutation Operator for Neuroevolution of Augmenting Topologies (NEAT)

Aldo Marzullo*, Claudio Stamile†‡, Giorgio Terracina*, Francesco Calimeri*, Sabine Van Huffel†‡
*Department of Mathematics and Computer Science
University of Calabria, Italy
Email: {marzullo, terracina, calimeri}@mat.unical.it
†Department of Electrical Engineering (ESAT), STADIUS
Katholieke Universiteit Leuven, Belgium
Email: {Claudio.Stamile, Sabine.Vanhuffel}@esat.kuleuven.be
‡imec
Leuven, Belgium

***Abstract*—In Genetic Algorithms, the *mutation operator* is used to maintain genetic diversity in the population throughout the evolutionary process. Various kinds of mutation may occur over time, typically depending on a fixed probability value called *mutation rate*. In this work we make use of a novel data-science approach in order to adaptively generate mutation rates for each locus to the Neuroevolution of Augmenting Topologies (NEAT) algorithm. The trail of high quality candidate solutions obtained during the search process is represented as a third-order tensor; factorization of such a tensor reveals the latent relationship between solutions, determining the mutation probability which is likely to yield improvement at each locus. The single pole balancing problem is used as case study to analyze the effectiveness of the proposed approach. Results show that the tensor approach improves the performance of the standard NEAT algorithm for the case study.**

***Keywords*-NEAT; Genetic Algorithm; Mutation; Tensor decomposition;**

I. Introduction

Neuro Evolution of Augmenting Topologies (NEAT) [17] is an algorithm developed by Kenneth O. Stanley and Ritso Miikkulainen at the University of Texas, used to optimize both the structure and weights of a neural network. NEAT aims at finding a solution to many technical challenges of the Topology and Weight Evolving Artificial Neural Networks problem: (i) provide a genetic representation that allows a meaningful application of the crossover operation, (ii) protect topological innovations avoiding the "premature" disappearance of a new network from the gene pool, so that it can be optimized, (iii) minimize topologies throughout evolution without the need for a specially contrived fitness function that measures complexity.

The NEAT genome structure contains a list of genes representing neurons, *neuron genes*, and connections, *link genes*. A link gene holds information about the two neurons it connects, the weight related to the connection, a flag that indicates if the link is enabled, a flag that indicates if the link is recurrent, and an *innovation number*. A neuron gene holds information about the type of the neuron (input, output or hidden) and the activation function.

The mutation operation in NEAT consists of several mutation operations that can be performed on the parent genome. In particular, four kinds of mutation can occur throughout the evolutionary process, which can modify both weight and structure: adding new connections, perturbing a connection weight, adding new hidden nodes, disabling or enabling genes in the chromosome. In the add-node mutation, an existing connection is split, and a node is placed where the old connection used to be; the old connection is disabled, and two new connections are added to the genome. The connection between the first node in the chain and the new node is given a weight of one, and the connection between the new node and the last node in the chain is given the same weight as the connection being split. Such a way of splitting the connection introduces a nonlinearity (i.e., sigmoid function) where there was none before. If initialized as described, the nonlinearity just slightly changes the function, and the new node is immediately integrated into the network. Old behaviors encoded in the preexisting network structure are not destroyed, and remain qualitatively the same, while the new structure provides an opportunity to elaborate on these original behaviors.

As in many standard genetic algorithms, in NEAT the probability for a mutation to occur is usually assigned as a constant predefined value, so that all chromosomes have the same likelihood of mutation, which does not take into account their fitness. In this work, we use tensor analysis in order to generate mutation probabilities for each locus of a chromosome which represents a candidate solution, thus following the idea previously shown in [1].

Tensorial techniques constitute powerful analysis methods for high dimensional data, widely used in data mining and machine learning applications, which have been proven to be able to reveal latent relationships among different dimensions [2]. In our approach, the trail of high quality solutions

978-1-5090-4601-0/17/$31.00 ©2017 IEEE

(i.e., neural network produced during the evolutionary process) is encoded as a third order tensor. Tensor factorization is then applied in order to reveal the latent relationship between various chromosome locations, thus identifying common subspaces of the solutions and producing mutation rates in such a way that mutation is more likely to generate better offspring. Moreover, the work is focussed on weight perturbation.

The remainder of the paper is structured as follows. In Section II we provide a detailed description of our approach, and in Section III we present our experimental activities. In Section IV we discuss our results, eventually drawing our conclusions in Section V.

II. Proposed Approach

In the following we describe the background techniques and methods, and provide further details on the proposed approach.

A. Neural network bi-dimensional representation

It is worth noting that the way a solution is represented is a crucial point for the tensor factorization, in order to reveal latent relationships among different individuals. The candidate solution (chromosome) of a generic problem, for NEAT, is a neural network, directly applied to the task in order to compute its fitness. In this context, a neural network is a collection of genes representing neurons and connections. Interestingly, such a neural network can be seen as a directed graph $G = (V, E)$, where V is a finite nonempty set of vertices, and E is a set of edges connecting ordered pairs of vertices in V. Here, vertices constitute input, hidden and output neurons, whereas edges constitute connections among neurons.

Adjacency matrix is a common way of representing a generic graph. The adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{i,j})$ such that

$$a_{i,j} = \begin{cases} 1 \text{ if} (i,j) \in E \\ 0 \text{ otherwise} \end{cases}$$

In the proposed approach, considering the above analogy, the solutions of the problem can be represented as set of matrices. Let $\mathcal{P} = \{S_1, \ldots, S_Q\}$ be a population of chromosomes obtained after a predefined number of generations, a specific solution $S_q = (V_q, E_q)$, $1 \leq q \leq Q$, can be represented as a matrix $\bar{S}_q \in \mathbb{R}^{N \times N}$ where $N = max(|V_1|, |V_2|, \ldots, |V_Q|)$. An element $\bar{s}^q_{i,j} \in \bar{S}_q$ is defined as follows:

$$\bar{s}^q_{i,j} = \begin{cases} \frac{1}{|E_q|} \text{ if} (i,j) \in E_q \\ 0 \text{ otherwise} \end{cases}$$

Roughly speaking the number of nodes in each matrix is defined by the maximum cardinality of neuron genes among all the individuals generated. If the number of nodes can change, the connections between nodes remain equal to the original.

Intuitively, each entry $\bar{s}^q_{i,j}$ of the matrix representing the solution, acts as an identifier for a given solution. Chromosomes with the same number of connections are encoded using the same value of $\bar{s}_{i,j}$, so that these individuals have a similar likelihood of mutation.

B. Tensorization

We denote as *tensor* a multidimensional array. Generally speaking, a N^{th}-$order$ tensor is an element of the tensor product of N vector spaces, each of which has its own coordinate system; for instance, a third-order tensor has three indices. A first-order tensor is a vector, a second-order tensor is a matrix, and tensors of order three or higher are called higher-order tensors.

In order to help the description of the tensor-based formalism herein employed, we next introduce some notation; for everything not explicitly mentioned in the following, we use the notation described in [11]. We denote scalar values with small letters (e.g., a), 1-dimensional vectors with bold small letters (e.g., $\boldsymbol{a}$), matrices with boldface capital letters (e.g., $\boldsymbol{A}$) and tensors with boldface Euler script letters (e.g., $\boldsymbol{\mathcal{A}}$).

Using the tensor formalism, it is possible to have a compact representation of a given (large or small) population of chromosomes. In more detail, in the proposed approach, a set of solutions is represented by a third-order tensor generated from the matrix representation of each chromosome (Section II-A).

As an example, the tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{N \times N \times R}$, illustrated in Figure 1, is built by stacking, in the third mode, the matrices representing the best $p\%$ (R) of the chromosomes. Where, p is an hyperparameter which should be found empirically during experiments. Individuals are inserted into the tensor in ascending order with respect to their fitness.

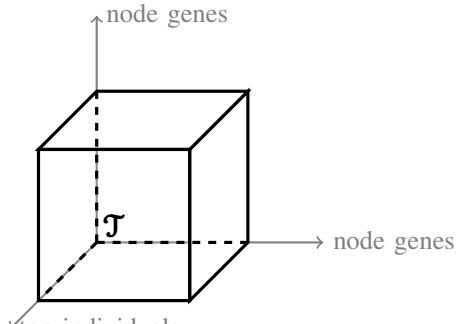


Figure 1. Tensor representation of the top individuals

C. Tensor factorization

Tensor factorization (decomposition) is a widely used method for identifying correlations and relations among

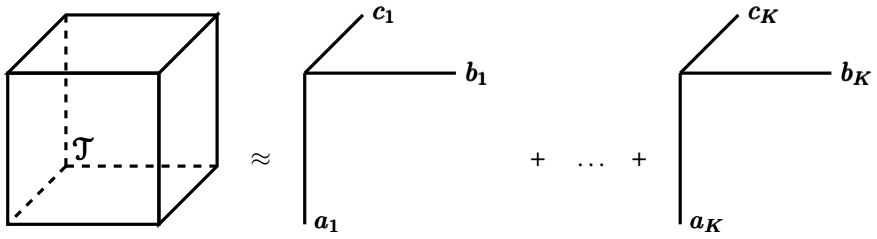


Figure 2. Graphic representation of the canonical polyadic decomposition (CPD).

different modes of high-dimensional data, which is applied in many research fields such as psychometrics, chemometrics, signal processing, numerical linear algebra, computer vision, numerical analysis, data mining, neuroscience, graph analysis, and more. The tensor decomposition methods are mainly generalizations of the Singular Value Decomposition (SVD) [7]. A number of different factorization methods have been proposed in the literature, such as Higher Order SVD (HOSVD) [12], Tucker decomposition [18], Parallel Factor (also known as PARAFAC or CANDECOMP or CP) [10] and Non-negative Tensor Factorization (NTF) [16]. In this study, we refer to the canonical polyadic decomposition (CPD) method [2].

CPD factorizes a tensor into a sum of component rank-one tensors. For example, given a third-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{i \times j \times z}$ we aim at writing it as:

$$\boldsymbol{\mathcal{X}} = \sum_{r=1}^{K} \boldsymbol{a_r} \circ \boldsymbol{b_r} \circ \boldsymbol{c_r} + \boldsymbol{\mathcal{E}} \tag{1}$$

where K is a positive integer, $\boldsymbol{a_r} \in \mathbb{R}^i, \boldsymbol{b_r} \in \mathbb{R}^j, \boldsymbol{c_r} \in \mathbb{R}^k \; \forall \, 1 \leq r \leq K$ are the component vectors and $\boldsymbol{\mathcal{E}} \in \mathbb{R}^{i \times j \times z}$ is the error tensor. The symbol "$\circ$" represents the vector outer product.

The rank of a tensor $\boldsymbol{\mathcal{X}}$, denoted $rank(\boldsymbol{\mathcal{X}})$, is defined as the smallest number of rank-one tensors that generate $\boldsymbol{\mathcal{X}}$ as their sum. In other words, this is the smallest number of components in an exact CP decomposition, where "exact" means that there is equality in equation 1 with the residual tensor $\boldsymbol{\mathcal{E}}$ as a zero-element tensor. An exact CP decomposition with $K = rank(\boldsymbol{\mathcal{X}})$ components is referred to as the rank decomposition.

The CPD problem can be formalized as follows:

$$\min_{\hat{\mathbf{x}}} \|\boldsymbol{\mathcal{X}} - \hat{\boldsymbol{\mathcal{X}}}\| \quad \text{with} \quad \hat{\boldsymbol{\mathcal{X}}} = \sum_{r=1}^{K} \boldsymbol{a_r} \circ \boldsymbol{b_r} \circ \boldsymbol{c_r} = [\![\boldsymbol{A}; \boldsymbol{B}; \boldsymbol{C}]\!] \tag{2}$$

where $\boldsymbol{A} \in \mathbb{R}^{i \times K}$, $\boldsymbol{B} \in \mathbb{R}^{j \times K}$, $\boldsymbol{B} \in \mathbb{R}^{z \times K}$.

The factorization is then obtained by solving the following optimization problem:

$$\min_{\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}} \|\boldsymbol{\mathcal{T}} - \boldsymbol{A} \circ \boldsymbol{B} \circ \boldsymbol{C}\| \tag{3}$$

The formulation of the tensor factorization using CPD is graphically illustrated in Figure 2.

Tensor decomposition allows to represent data in a more concise and generalizable manner that prevents missing data and other anomalies. Furthermore, tensor factorization methods feature several interesting properties that turn out to be useful in many contexts. Indeed, factorization helps at partitioning data into a set of more comprehensible sub-data, which can be of specific use according to various criteria and the application at hand. In the context of Genetic Algorithms, for instance, the search history of the generated solutions can be turned into a multi-dimensional array, representing the individuals that change during time.

D. Method description

In this work, even though the standard NEAT implementation is used, we substituted the generic mutation operator with a tensor-based approach; this allowed us to improve performances via an adaptive locus based mutation operator. In the first phases of the evolution, a random population of neural networks is generated. During the evolution process, crossover and mutation operations are applied in order to produce better offsprings. Evolution continues until the target fitness is reached, or the stopping criterion is met. Our approach modifies the procedure similarly to what is presented in [1]. Every η generations, a third-order tensor $\boldsymbol{\mathcal{T}}$ is created, which encodes the $p\%$ of the best generated chromosomes, as already introduced in Section II-B. Then, the tensor is factorized in its basic factors, thus producing a basic frame; each entry of such basic frame is used as the mutation probability for the connection genes in the next η generations. In more detail, the tensor $\boldsymbol{\mathcal{T}}$ is factorized in its basic factors, with $K = 1$, producing the approximation $\hat{\boldsymbol{\mathcal{T}}}$ as described by the following equation:

$$\hat{\boldsymbol{\mathcal{T}}} = \boldsymbol{a} \circ \boldsymbol{b} \circ \boldsymbol{c}$$

where the length of the vectors $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$ are $|N|, |N|$ and R respectively.

The outer product between $\boldsymbol{a}$ and $\boldsymbol{b}$ produces a *Basic Frame* ($\boldsymbol{F}$) [1] which has size $|N| \times |N|$, i.e., the same size of a generic neural network encoding (in particular, the basic frame contains real values between 0 and 1). As illustrated in [1], these values point towards regions in encoded networks where change of entry values has been a common pattern among good quality matrices. Each entry in

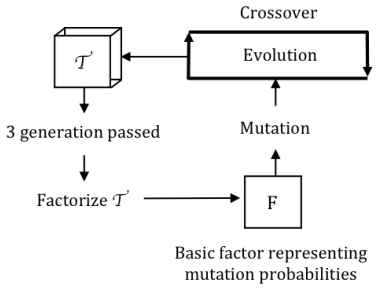


Figure 3. NEAT + tensor decomposition framework

the basic frame can be then used as a mutation rate of each locus for the next generations, until a new tensor is created. Indeed, during the next η generations, the connection i, j is mutated with probability $F_{i,j}$, where i, j indicates input and output neuron, respectively, of a generic chromosome.

The pseudocode of the algorithms used to create and factorize the tensor, and the one that performs the application of the basic frame to the mutation operator of NEAT, are reported in Algorithm 1 and Algorithm 2, respectively. In Algorithm 1, during the evolution, the chromosome producing the best fitness for each generation is added to a (sorted) list. After a certain number of generations, the trail of high-quality solutions is extracted and used to create the basic frame. In Algorithm 2, the classic NEAT mutation operator is modified in such a way that each connection gene is mutated with a probability depending on the value contained in the entry of the basic frame, identified by the input id and the output neurons of the connection. Note that our approach focuses on weight perturbation only. The whole process is illustrated in Figure 3.

Algorithm 1: Basic Frame Construction

Function *construct_F(*η, K, p, curr_best, bests, F, gen*)*
- **Input:** η number of generation to compute F
- **Input:** K: the rank of the tensor
- **Input:** p: the percentage of best chromosomes
- **Input:** bests: sorted list of the best chromosomes
- **Input:** curr_best: current generation best chromosome
- **Input:** F: the basic frame used during the η generation
- **Input:** gen: generation number
- **Output:** F: the basic frame
- **if** gen $=\eta$ **then**
  - $r \leftarrow$ `length(`curr_best`)` $*$ p$/100$;
  - $\mathcal{X} \leftarrow$ curr_best$[r:]$;
  - $a, b, c \leftarrow$ `factorize(`$\mathcal{X}$, K`)`;
  - F $\leftarrow a \circ b$;
  - curr_best $\leftarrow \{\}$; return F;
- **end**
- `insertOrdered(`curr_best, bests`)`;
- return F;

Algorithm 2: Mutation Operator

Function *mutate(F)*
- **Input:** F is the basic frame previously constructed
- **for** connection $\in$ connection_genes **do**
  - $in \leftarrow$ connection.node_in;
  - $out \leftarrow$ connection.node_out;
  - $size \leftarrow$ `length(`F`)`;
  - **if** $in < size \wedge out < size$ **then**
    - **if** `random()` $< F_{in,out}$ **then**
      - `mutate_connection(`connection`)`;
    - **end**
  - **end**
- **end**

III. Experiments

In this section we report the details of our experimental activity and we discuss the results, comparing the proposed approach with the classic implementation of NEAT. We preliminarily provide the reader with a brief description of the pole balancing problem used in the experiments.

A. The pole balancing problem

The pole balancing problem is a classic problem in dynamics and control theory, and is widely used as a benchmark for testing control algorithms and artificial learning systems [5], [4], [3], [14], [15]. A system must be controlled, consisting of a pole (more specifically, an inverted pendulum) that is connected to a motor-driven cart positioned within some track boundaries. A certain force can be applied to the cart in the x direction at regular time intervals, with the aim of keeping the pole balanced indefinitely, and making the cart stay within the track boundaries. The state of such system is defined by four variables: the angle of the pole from vertical in the x direction θ_x, the angular velocity of the pole $\overline{\theta_x}$, the position of the cart in the plane x, and the velocity of the cart $\overline{x}$ [8]. In a typical implementation of the controller, in our case a neural network, it receives as input information about the system at each time step (i.e., the positions of the poles, their respective velocities, the position and velocity of the cart, etc). A trial typically begins with the pole off-center by a certain number of degrees; the controller moves the cart in either direction, avoiding it to fall off either edges of the track. A more complex formulation of the problem involves two poles, both fixed on the same point on the cart [8].

B. Experimental setting

For the experiments, all parameters defined in the implementation provided by the framework[1] we used for testing have been maintained, provided that the same configuration was also adopted by other publicity available implementations. In more detail, population size is fixed at 250 individuals, and the maximum fitness threshold is 60000. Crossover and mutation operators are uniform and traditional, respectively. Probability of adding and removing connections are set to 0.1 and 0.05, whereas probability of

[1]https://github.com/CodeReclaimers/neat-python

Table I
COMPARISON OF THE PROPOSED APPROACH AND THE CLASSIC NEAT ALGORITHM ON THE POLE BALANCING PROBLEM OVER 20 RUNS. SYMBOLS $\gg$ AND $>$ INDICATE RESPECTIVELY A BIG (GREATER THAN 10) AND A SMALL (LESS THAN OR EQUAL TO 10) DIFFERENCE BETWEEN THE APPROACHES.

	CLASSIC			ADAPTIVE	
Run	Fitness	Generations	vs	Fitness	Generations
1	60000	56	$\gg$	60000	28
2	326	100	$\gg$	60000	26
3	60000	61	$\ll$	2459	100
4	264	100	$\gg$	60000	21
5	60000	79	$\ll$	60000	38
6	628	100	$\gg$	60000	45
7	60000	90	$<$	4952	100
8	60000	61	$>$	60000	52
9	5276	100	$\gg$	60000	25
10	208	100	$\gg$	60000	12
11	2955	100	$\gg$	60000	34
12	60000	49	$\gg$	60000	19
13	1807	100	$\gg$	60000	58
14	738	100	$\gg$	60000	31
15	1623	100	$\gg$	60000	24
16	60000	55	$<$	60000	67
17	169	100	$\gg$	60000	26
18	60000	70	$\gg$	60000	25
19	60000	39	$\ll$	60000	57
20	960	100	$\gg$	60000	43

Table II
SUMMARY COMPARISON OF THE PROPOSED APPROACH AND THE CLASSIC NEAT ALGORITHM OVER 20 AND 100 RUNS

	CLASSIC		ADAPTIVE	
Runs	AVG Fitness	AVG Generations	AVG Fitness	AVG Generations
20	27747.7	83.0 ($\pm$21.7)	54370.6	41.6 ($\pm$24.8)
100	28873.9	86.9 ($\pm$19.9)	40681.4	69.5 ($\pm$28.3)

adding and deleting nodes are set to 0.1 and 0.05, respectively. Probability of mutating weights is 0.8 and probability of replacing weights is 0.1. Compatibility threshold is 3.0, and the values for excess and disjoint coefficient are set to 1.0. We used $\eta = 3$, $p = 28$ in our settings.

Throughout the experiments, our approach is referred to as the *adaptive* one, whereas the original method is referred to as the *classic* one. Two tests using 20 and 100 runs were performed, using both *classic* and *adaptive* method, where each run has been applied on a different input. Analysis in terms of quality of the solution (fitness value) and speed of the convergence (number of generations needed to reach the best fitness value) have been evaluated. Different runs were performed, and the corresponding results statistically compared: indeed, Wilcoxon-Mann-Whitney test [19], [13] was conducted to asses statistical differences between the two methods. The test were computed with a level of significance of 5%. Numerical results are presented as average $\pm$ their standard deviation (SD).

Table III
EXECUTION TIME COMPARISON OF THE PROPOSED APPROACH AND THE CLASSIC NEAT ALGORITHM OVER 10 TESTS OF 100 GENERATIONS EACH

Run	Classic	Adaptive
1	4m7.105s	5m53.625s
2	2m35.905s	3m26.999s
3	2m41.799s	3m28.474s
4	4m10.004s	15m41.834s
5	2m52.639s	4m1.610s
6	2m31.425s	2m53.777s
7	4m17.594s	3m30.016s
8	3m16.992s	7m43.061s
9	3m52.969s	13m18.292s
10	4m46.159s	6m22.795s

IV. DISCUSSION

Complete results for the experiments performed using 20 generations are reported in table I. The *adaptive* version of NEAT algorithm clearly outperforms the *classic* version: in 14 different runs (out of 20), the *adaptive* method reaches better performances in terms of both numbers of generations and fitness.

Compact results for the experiments performed using 20 and 100 runs are reported in table II. In both cases, the *adaptive* method outperforms the *classic* one. Indeed, over 100 runs, the average fitness for the *classic* method is 28873.9, whereas it reaches 40681.4 for the *adaptive* one. A similar scenario can be depicted for the number of needed generations: 86.9 and 69.5 for *classic* and *adaptive* versions, respectively.

The results obtained by our *adaptive* method are also significant. Indeed, with 20 runs, we found a p-value < 0.01 for the fitness function and < 0.001 for the generations. With 100 runs, we obtained a p-value < 0.001 for both fitness function and number of generations.

As far as execution time is concerned, our experiments show that our approach is slightly more time demanding with respect to the classical implementation. Table III, reports a comparison of the execution time needed for running 100 generations. Except for some cases, we can observe that the difference is in the order of one minute. However, the

Table IV
EXECUTION TIME AND CONVERGENCE COMPARISON

	CLASSIC		ADAPTIVE	
Run	Generations	Time	Generations	Time
1	99	2m28.922s	43	2m7.448s
2	99	2m37.509s	99	5m14.943s
3	72	2m8.442s	18	0m46.888s
4	82	2m15.340s	99	15m35.550s
5	80	2m15.880s	99	4m17.291s
6	99	2m27.518s	48	2m48.337s
7	99	3m16.512s	58	2m11.577s
8	99	2m25.164s	44	1m46.004s
9	99	2m33.700s	99	5m15.031s
10	82	2m16.025s	25	1m1.251s

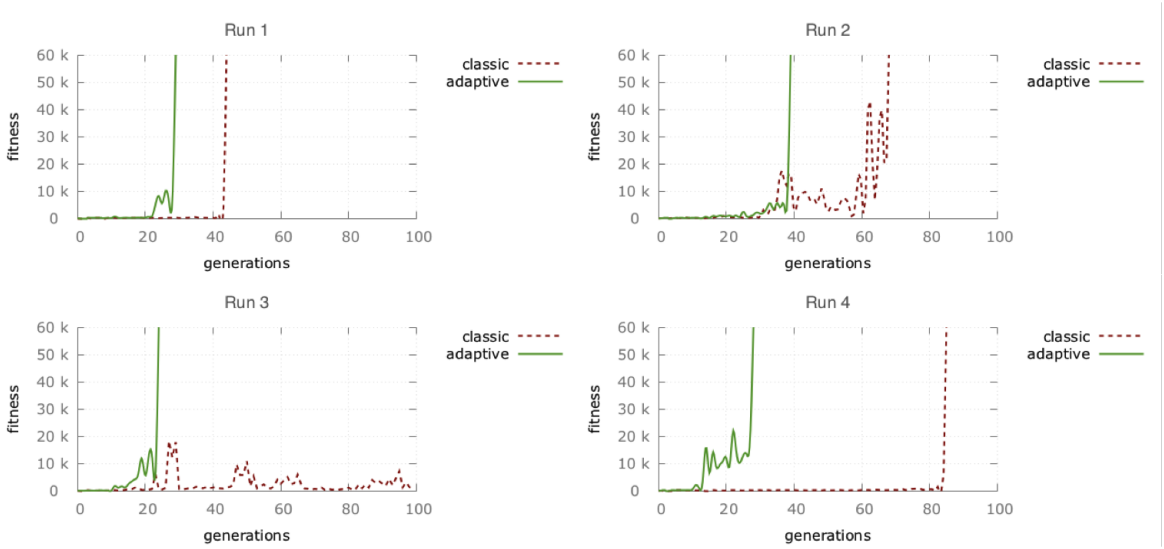


Figure 4. Graphical visualization of four sample runs

extra-time needed on a per-generation basis, is generally compensated by a faster convergence and, consequently, by a possibly faster running time, as can be observed from Table IV.

We finally report in Figure 4 some graphs showing the evolution of the fitness function for the first four runs of our tests. These figures show not only that the *adaptive* version of the algorithm allows to reach the highest level of the fitness function, as already illustrated in Table I, but also that it is able to reach the final level in much fewer steps than the *classic* one. Observe that we used the same initial random population as seed for both the classic and adaptive version in order to avoid bias in the initialization step. Presented graphs are representative of the overall results obtained throughout the experiments.

V. Conclusion

In this study we proposed a new mutation operator based on tensor factorization within the NEAT algorithm. We formalize the NEAT neural network as a matrix; this compact representation allowed us to create a third-order tensor composed by the trail of the best individuals generated during a fixed number of generations.

Canonical polyadic decomposition was then applied to the generated tensor in order to obtain specific mutation probabilities for each connection gene.

Finally, in order to validate the method, we tested it against the pole balancing problem. Results of the experimental activities showed that the proposed approach achieves notably higher levels of performances, in the case study, when compared to the classic mutation operator used in the classic NEAT algorithm.

It is worth noting that the proposed approach considers just one of the possible mutations defined in NEAT; with this respect, interesting perspectives can arise if the tensor analysis is applied to other mutation operations. For instance, create a mutation operator to perturb the structure of the network, i.e., adding and removing neurons, or enabling or disabling connections. We also plan to apply the proposed approach to complex tasks related to medical imaging analysis.

Acknowledgments

Claudio Stamile has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Advanced Grant: BIOTENSORS (n 339804). EU MC ITN TRANSACT 2012 (n 316679). Francesco Calimeri has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 690974 for the project: "MIREL: MIning and REasoning with Legal texts".

References

[1] S. Asta, E. Özcan. "A Tensor Analysis Improved Genetic Algorithm for Online Bin Packing," Conference on Genetic and Evolutionary Computation (GECCO '15), Sara Silva (Ed.). ACM, New York, NY, USA, pp. 799-806, 2015.

[2] S. Asta, E. Özcan. "A tensor-based selection hyper-heuristic for cross-domain heuristic search," Information Sciences, vol. 299(0), pp. 412-432, 2015.

[3] C.W. Anderson. "Learning to control an inverted pendulum using neural networks," IEEE Control Systems Magazine, vol 9, pp. 31-37, 1989.

[4] R.S. Bapi, B. D'Cruz, G. Bugmann, "Neuro-resistive grid approach to trainable controllers: A pole balancing example," Neural Computing and Applications, vol. 5(1), pp. 33-44, 1997.

[5] J. Brownlee. “The pole balancing problem. A Benchmark Control Theory Problem,” Swinburne University of Technology, 2005.

[6] J. Carroll, J. Chang. “Analysis of individual differences in multidimensional scaling via an n-way generalization of Eckart-Young decomposition,” Psychometrika, vol. 35(3), pp. 283-319, 1970.

[7] G.H. Golub, C. Reinsch. “Singular value decomposition and least squares solutions,” Numerische mathematik, vol. 14(5), pp. 403-420, Springer, 1970.

[8] F. Gomez, R. Miikkulainen. “2-D Pole Balancing with Recurrent Evolutionary Networks,” International Conference on Artificial Neural Networks (ICANN98), New York: Elsevier, 1998.

[9] R.A. Harshman. “Foundations of the PARAFAC procedure: Models and conditions for an explanatory multi-modal factor analysis,” UCLA Working Papers in Phonetics, vol. 16(1) p. 84, 1970.

[10] R.A. Harshman. “Methods of three-way factor analysis and multidimensional scaling according to the principle of proportional profiles,” PhD thesis, University of California, Los Angeles, CA, 1976.

[11] T.G. Kolda, B. W. Bader, “Tensor Decompositions and Applications,” SIAM Review, vol. 51(3), pp. 455-500, 2009.

[12] L.D. Lathauwer, B. D. Moor, J. Vandewalle. “A multilinear singular value decomposition,” SIAM J. Matrix Anal. Appl, vol. 21(4), 1253-1278, 2000.

[13] H.B. Mann, D.R. Whitney. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other,” Annals of Mathematical Statistics. vol. 18 (1): pp. 50-60, 1947.

[14] D. Michie, R. A. Chambers. “BOXES: An experiment in adaptive control,” E. Dale and D. Michie, editors, Machine Intelligence. Oliver and Boyd, Edinburgh, UK, 1968.

[15] J. Schaffer, R. Cannon. “On the control of unstable mechanical systems,” In Automatic and Remote Control III: Proceedings of the Third Congress of the International. Federation of Automatic Control, 1966.

[16] A. Shashua, T. Hazan. “Non-negative tensor factorization with applications to statistics and computer vision,” In ICML, pp. 792-799, 2005.

[17] K. O. Stanley, R. Miikkulainen. “Evolving Neural Networks Through Augmenting Topologies,” Evolutionary Computation, vol. 10(2), pp. 99-127, 2002.

[18] L. R. Tucker. “Some mathematical notes on three-mode factor analysis,” Psychometrika, vol. 31, pp. 279-311, 1966.

[19] F. Wilcoxon. “Individual comparisons by ranking methods,” Biometrics bulletin, vol. 1(6), pp. 80-83, 1945.