Evolution of the Topology and the Weights of Neural Networks using Genetic Programming with a Dual Representation

João Carlos Figueira Pujol and Riccardo Poli

School of Computer Science
The University of Birmingham
Birmingham B15 2TT, UK
E-MAIL: {J.Pujol,R.Poli}@cs.bham.ac.uk

Technical Report CSRP-97-7

February 6, 1997

Abstract

Genetic programming is a methodology for program development, consisting of a special form of genetic algorithm capable of handling parse trees representing programs, that has been successfully applied to a variety of problems. In this paper a new approach to the construction of neural networks based on genetic programming is presented. A linear chromosome is combined to a graph representation of the network and new operators are introduced, which allow the evolution of the architecture and the weights simultaneously without the need of local weight optimization. This paper describes the approach, the operators and reports results of the application of the model to several binary classification problems.

1 Introduction

The design of neural networks is still largely performed using lengthy process of trial and error definition of the topology, followed by the application of a learning algorithm such as backpropagation [1]. However, the literature describes some approaches which try to automatically determine the topology and the weights simultaneously, by deleting or adding structural elements (neurons and connections) to an existent unsatisfactory architecture. Destructive approaches (see for example [2]) begin with a trained architecture richer than necessary and prune unimportant elements of the network, whereas constructive approaches [3, 4, 5, 6] begin with a small topology and add new elements as need arises during the training process. The disadvantage of these strategies is that they constrain the architecture of the networks evolved, either from the beginning, or through the structural modifications they introduce. For example, Fahlman [4] adds hidden neurons fully connected to existing ones in a feedforward architecture, Frean [3] uses threshold activation functions and adds new hidden neurons fully connected to the input layer, and Campbell *et al.* [6] use binary weights.

Recently, new promising approaches based on evolutionary computation have been proposed for the development of artificial neural networks.

Evolutionary computation is a class of global search techniques based on the learning process of a population of individuals [7, 8, 9, 10, 11, 12], each individual representing a potential solution to a given problem. Typical evolutionary algorithms update this population seeking for better regions of the search space using operations of selection, recombination and mutation, inspired by biological evolution.

Many methods perform separately the building of the architecture and the learning of the weights [13, 14, 15, 16, 17]. Although biologically plausible, this approach is very inefficient. Other methods include partial training by gradient descent or simulated annealing in the evolution process [18, 19], and transfer the learned weights to the offspring in a Lamarckian fashion, in order to speed up evolution. The problem with these methods is that the performance of a partially trained network with a good architecture and bad weights may be worse than the performance of a bad architecture but a good initial set of weights. This can mislead the evolutionary process.

Other methods try to develop the weights and the architecture concurrently, but impose restrictions on the value of the weights [20, 21, 22, 23]. This problem has been reduced using evolutionary programming techniques [24, 19, 18, 25], which rely exclusively on mutation to adjust the weights and the architecture at each generation. However, this approach misses the benefits of the exchange of genetic material [9].

An alternative approach to the development of neural networks is to use genetic algorithms [26, 9] and genetic programming(GP)[27]. They are a class of the evolutionary algorithms which make large use of crossover as an evolution operator. Genetic programming, originally developed to evolve computer programs, uses parse trees to represent neural networks. [27, 28, 29].

However, these methods have been largely limited by the lack of good approaches to evolve the weights and the fact that parse trees are not a natural representation for neural networks. Indirect methods such as cellular encoding [21, 30] avoid the problem of representing the network directly as a parse tree, by representing the rules to construct the network instead. However, there are still constraints on the weights evolved, and to construct the network from the rules at each generation can be a considerable overhead.

Recently, a new form of GP, Parallel Distributed Genetic Programming (PDGP), in which programs are represented as graphs, has been applied to the development of neural networks with promising results [31, 32]. In PDGP, a two-dimensional grid is associated with the network to represent its layers. The genetic operators are specialized to act on this grid. However, no operators specialized in handling the connection weights were defined in PDGP, which also presented some representational biases, limiting its applicability to neural networks.

In this paper, we build upon the preliminary work done with PDGP, by introducing a dual representation and new operators. The graph representation of the network based on a two-dimensional grid is not static, but created dynamically on demand by some of the genetic operators, whereas other operators use a linear representation of the network. As this is more compact, we use it as the main representation. All operators are specialized to handle neural networks.

The new approach has been successfully applied to feedforward networks, and allows the determination of the architecture and the weights concurrently. Moreover, the use of genetic programming introduces interesting possibilities to be explored, which are impossible with other methods.

In the following sections, the representation and the operators are described, and the results of the application of this paradigm to binary classification problems are presented.

2 Dual representation

In (PDGP), instead of the usual parse tree representation, a graph representation was used, where the functions and terminals are allocated in a two-dimensional grid of fixed size and form built in the chromosome. The grid is particularly useful to evolve solutions to problems, like neural networks, which have a natural layered structure, as it allows the definition of genetic operators which preserve desired properties of the network. However, there is no need to build the grid in the chromosome, as it is the same for all individuals of the population. It is more natural to use a linear genotype and a separate grid description.

The linear representation includes every detail necessary to build and use the network: connectivity, weights and the activation function of each node. All chromosomes in the population have the same number of nodes. Like in the standard genetic programming, there is a set of functions, the admissible activation functions, and a set of terminals, the variables containing the input to the network.

The genotype is represented by a list of nodes (units). Each node may contain either a terminal or a function. In the first case, the output of the node is the value of the terminal. In the second case, the node contains not only the activation function, but also the connections and the weights (see Figure 1), to allow the computation of its output.

An index indicates the position of the nodes in the chromosome. The first nodes represent the input neurons, whereas the last ones represent the output neurons of the network, and their number depends on the problem to be solved. The remaining nodes are hidden. Note that the hidden nodes may also contain terminals, introduced either initially

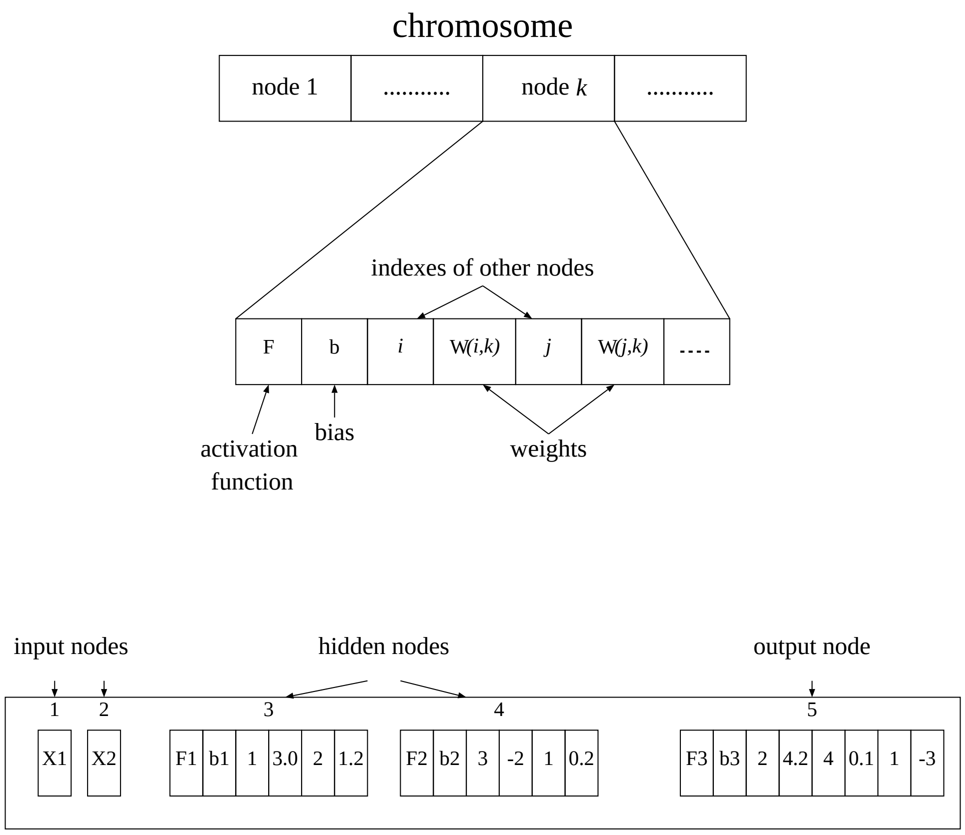


Figure 1: Function node description and an example of chromosome with 5 nodes.

or by the genetic operators, as described in the next section. The current implementation is restricted to feedforward networks, i. e. a node may only be connected to previous nodes within the chromosome.

Figure 2 shows an example of a network with two inputs, one hidden neuron and one output neuron, a common topology for the solution of the XOR problem. In the representation, however, there are 2 hidden units, one function node and a terminal, resulting in a five neuron network after the decoding of the genotype. The order of the connection declarations within a node description is irrelevant, and multiple connections to the same node are allowed (this feature is used to adjust the weights, as described later).

In standard genetic programming the size of the chromosomes (parse trees) may grow excessively. The linear representation avoids this problem constraining all chromosomes to having the same number of nodes.

The linear chromosome is associated to a grid, in which each node occupies a cell. This allows the application of constraints to the operators, which preserve desirable properties of the network. For example, a population of networks may be initialized with no connections within the same layer or between non-adjacent layers, and the grid may be used to guide the operators to comply with this pattern of connectivity if desired.

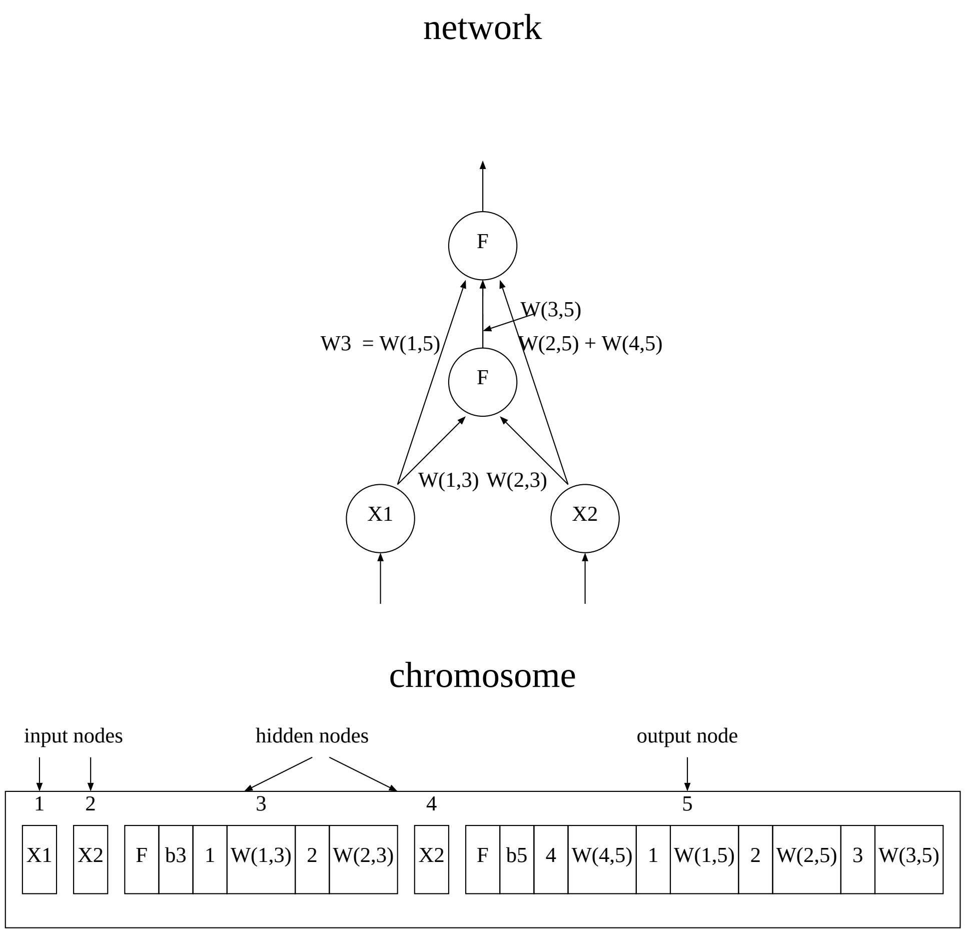


Figure 2: Network to solve the XOR problem and the corresponding chromosome.

The structure of the grid is defined by a description table (see Figure 3). The grid description table allows the construction of a lookup table, to map the nodes from the linear chromosome onto their grid positions. The grid-based representation is very similar to the phenotype, but more efficient to handle during crossover, because of its regularity. Moreover, crossover on the grid is very similar to crossover on graphs, but more efficient. Note that the grid is not necessarily rectangular, it may have any number of layers, each of different size. This may be used to constrain the geometric form of the network (e.g. to evolve encoder/decoders).

There is a large flexibility in the definition of the genetic operators within the framework of the new paradigm. A few of them are discussed in the next section.

3 Operators

3.1 Crossover

Crossover in genetic programming is performed by swapping subtrees between parents. In the dual representation used here, the operations may be performed swapping sub-graphs, or node description between parents. To define some of the operators a *node1* and a *node2* are randomly selected in the first parent and in the second parent, respectively. The operators are described as follows:

- *Layered sub-graph crossover.* A randomly selected sub-graph from the second parent is transferred into the first parent according to the two-dimensional grid. First, the sub-graph is defined by all the nodes connected to *node2* backward in the chromosome, and *node2* is transferred to the first parent at the position of *node1.* Note that the complete node is transferred, not only the weights. This means a node containing a terminal may replace a node containing a function, and vice-versa. In the grid, *node1* and *node2* define a vertical and a horizontal displacement. The transfer of the other nodes of the sub-graph to their new positions in the first parent is performed according to such displacements (see Figure 4a). If the vertical displacement of a node results in a position below the first layer of the grid (an infeasible connection), the connection is deleted. If the horizontal displacement results in a position outside the limits of the corresponding layer, the position of the node is wrapped around. This operation preserves the structure of the network as defined initially, in the sense that if there are no connections between non-adjacent layers or within a layer before the operation, there will be none after the operation either (see Figure 4a). The only exception is when a terminal replaces a function, creating a connection to the input layer in the phenotype. This crossover operator is similar to the *SANN* crossover in PDGP [31].

- *Linear sub-graph crossover.* First a linear displacement is defined by the difference between the positions of *node1* and *node2* within the linear chromosome. Secondly, *node2* is transferred to the first parent at the position of *node1.* Subsequently, the units connected to *node2* are transferred to their new positions in the first parent according to the linear displacement defined. Connections corresponding to node

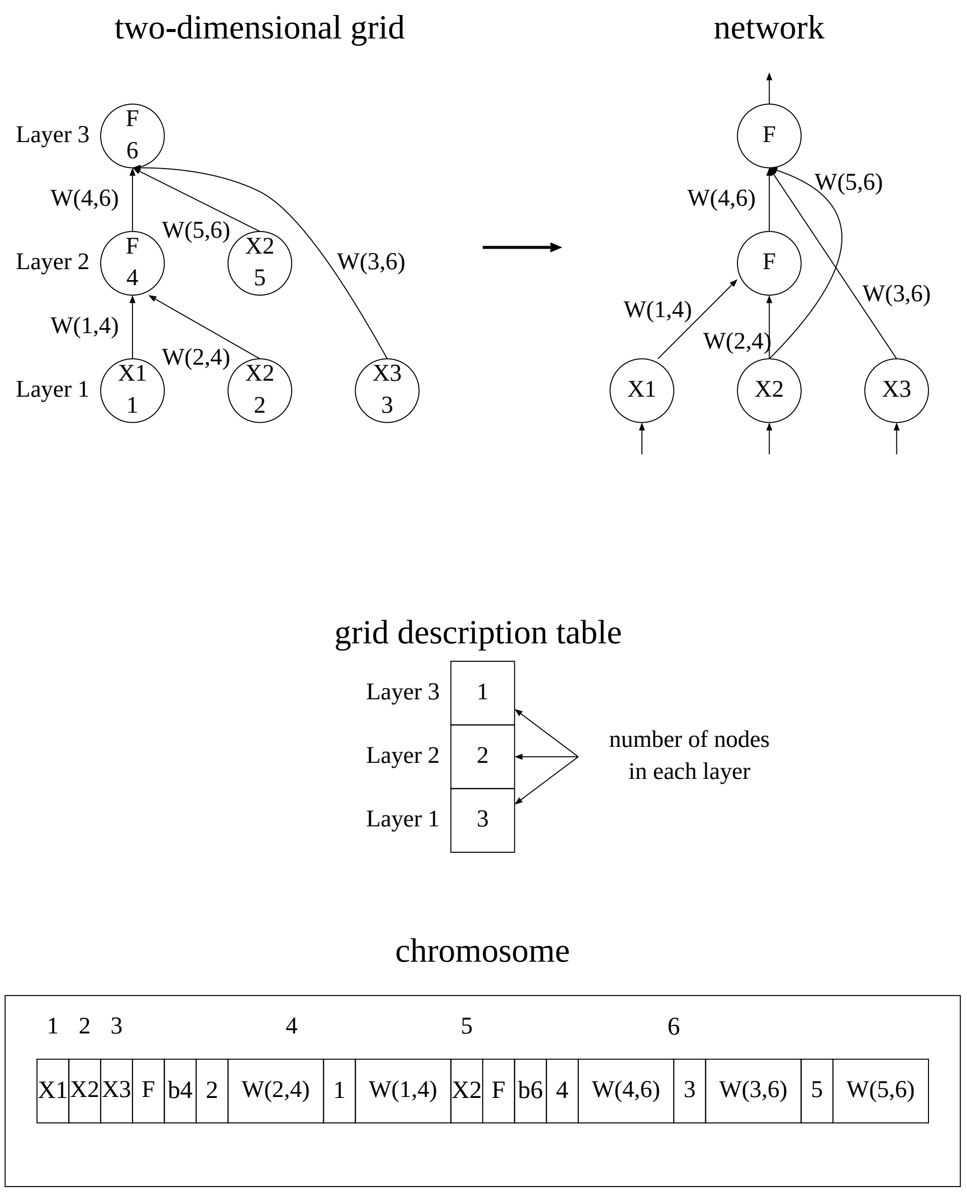


Figure 3: A chromosome, the associated grid and the decoded neural network (for simplicity, the biases were not indicated in the grid and in the network).

positions before the first unit are eliminated. This operation may create connections within the same layer (see Figure 4b).

- *Node transfer crossover.* Instead of transferring the sub-graph defined by the nodes connected to *node2*, only *node2* is transferred. As in the previous operations, infeasible connections are deleted.

- *One-point crossover.* Another possibility is to take advantage of the linear structure of the chromosome and perform one-point crossover as in a standard genetic algorithm, by randomly cutting the parents in a common point and swapping the left-hand sides to produce the offspring, with no regard to the connectivity.

- *Connection preserving crossover.* This can be used with the aforementioned crossover operators. When a node is transferred, infeasible connections are kept in their original place. This operation does not preserve the initial structure of the network, but may be useful to preserve the information contained in the connection weights, which would be lost otherwise.

- *Node description crossover.* In this case, *node1* and *node2* are cut at random points and an offspring is generated connecting the first part of *node1* to the second part of *node2* or vice-versa. This is a way not only of introducing new connections into a node, but also of adjusting the values of the weights, as multiple connections between two nodes may be created. After the application of the operator, multiple weights are added and multiple links grouped back into a unique connection (see Figure 5).

The idea behind sub-graph crossovers is that connected sub-graphs are functional units (building blocks) whose output is used by other functional units. Therefore, by replacing a sub-graph by another, different ways of combining building blocks are explored. The other crossover operators should be seen as methods to change the features of the building blocks, and to fine tune them.

Although many other operators may be defined, the basic crossover operators presented here give an idea of the possibilities offered by our dual representation.

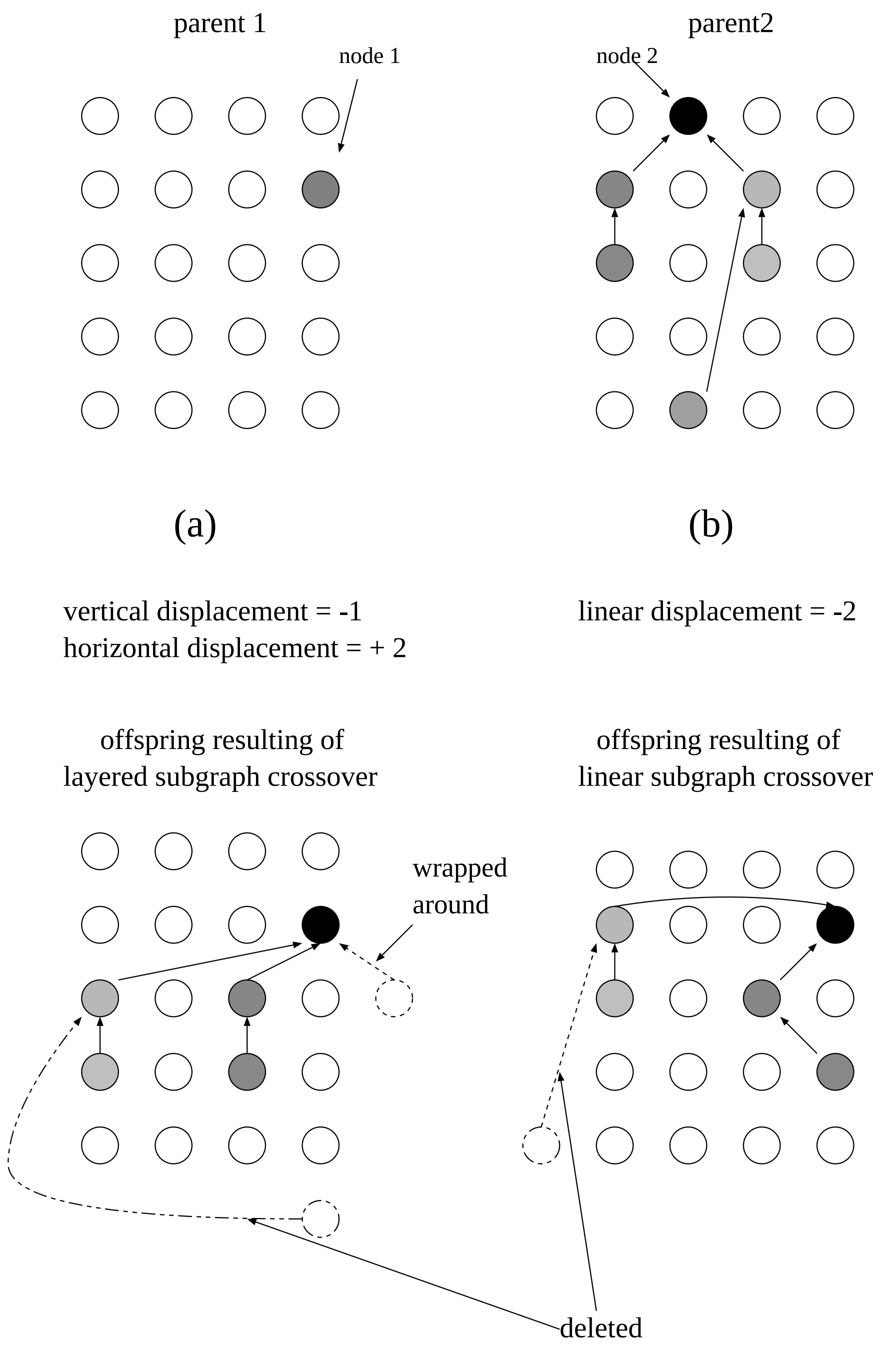


Figure 4: Crossover by sub-graph transfer. The relevant connections and nodes are indicated with different patterns to make easier to track them to their new positions. Dotted connections exceed the limits of the grid. In the case of layered crossover, one of them was wrapped around and the other one was eliminated.

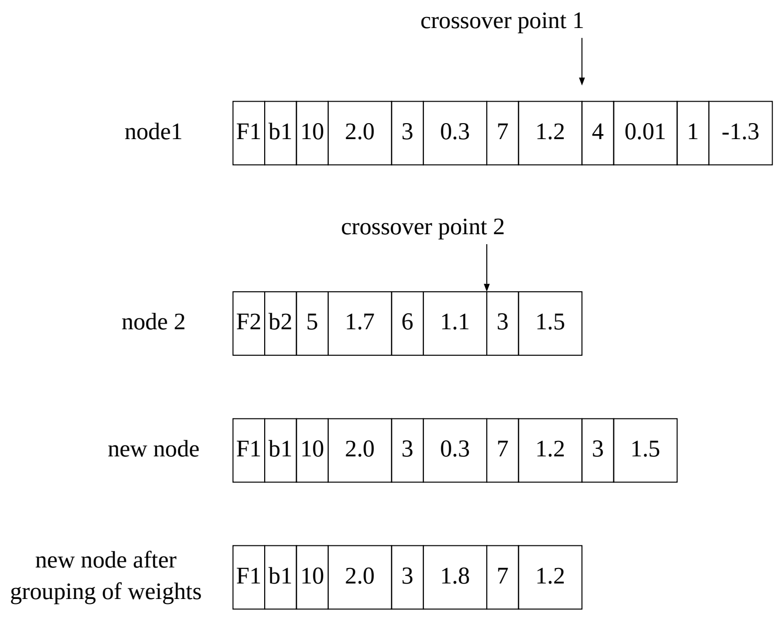

crossover point 1
node1
F1 b1 10 2.0 3 0.3 7 1.2 4 0.01 1 -1.3
crossover point 2
node 2
F2 b2 5 1.7 6 1.1 3 1.5
new node
F1 b1 10 2.0 3 0.3 7 1.2 3 1.5
new node after grouping of weights
F1 b1 10 2.0 3 1.8 7 1.2

Figure 5: Node description crossover creates new connections and adjusts weights.

3.2 Mutation

The dual representation also allows the implementation of the whole set of mutation operators associated with evolutionary methods applied to neural networks: addition and deletion of connections, crossover of a chromosome with a randomly generated one, crossover of a randomly selected node with a randomly generated one, addition of Gaussian noise to the weights and biases, etc..

The deletion or addition of nodes is not allowed in our representation, as the size of the chromosomes is constant. However, a hidden function node may be replaced by a terminal node or vice-versa, and this may be used to reduce or increase the complexity of the network, within predefined limits.

4 Pruning the network

In order to find solutions with minimum complexity, the following strategy is applied in our GP runs: after a 100% correct solution is found, we replace one hidden node of each chromosome in the population by a terminal, and the evolution process is resumed. This pruning procedure is repeated until the specified number of generations is achieved. This strategy has the advantage of generating solutions of varying degrees of complexity, allowing the user to decide which solution is preferable: a complex one, possibly presenting fault tolerance, or a parsimonious one, with probably more generalization power. No penalty term is included in the fitness function to enforce more parsimonious solutions.

5 Results

In all experiments a population of 200 chromosomes was evolved for a maximum of 200 generations. For each problem, we performed 100 runs with different random seeds. In the experiments, we used a generational genetic algorithm with: tournament selection (tournament size = 4) and crossover and mutation probabilities of 70% and 5%, respectively.

Unless otherwise stated, all chromosomes were initialized with 10 hidden function nodes (no terminals were present in the hidden layer initially) and random connections. The weights and biases were randomly initialized within the range [-1,1].

The only mutation operator used was the addition of zero mean Gaussian noise (with constant standard deviation 0.05) to the weights.

The fitness function was the standard mean square error of the output of the network for all input patterns. The function set included a threshold activation function.

The method was applied to N-bit odd parity problems (the network output must return 1 if there is an odd number of 1's in the input pattern, and 0 otherwise), with N=2 (XOR), 3 and 4. Minimal solutions were found in all problems.

In the experiments *node description crossover* was combined to each of the other crossover operators with a relative probability of 3 and 1, respectively. The *node description crossover* was used to fine tune the weights, whereas the other ones were responsible for distributing the weights and determining the architecture of the network.

Tables 1, 2, 3 and 4 show the results obtained. Column 2 represents the average number of generations to obtain the last solution through the pruning process described. Column 3 and 4 show the number of hidden nodes and the number of connections after applying the pruning strategy described, respectively. Column 5 shows the computational effort, i. e. the number of fitness evaluations, necessary to obtain a solution with 99% probability [27], and column 6 shows the percentage of runs in which solutions were found. (Averages over the successful runs.)

Apparently, *node transfer crossover* gives the best results in terms of effort and percentage of solutions found. However, Table 4 shows better results for *one-point crossover* applied to the XOR problem. Taking account only the average number of generations to achieve minimum solutions, *one-point crossover* seems to be more effective in all tasks.

The grid representation of a typical solution for the XOR problem and the corresponding network are shown in Figure 6a. The 11th function node without output connections and the disconnected nodes in the hidden layer are introns, which are eliminated by the interpreter when decoding the chromosome. The interpreter also replaces connections departing from terminals in the hidden layer with connections departing from terminals in the first layer. Similar results are shown for the 3 and 4 parity problems in Figures 6(b) and 6(c), respectively. As expected, the pruning process tends to generate solutions with connections between the first layer and the output layer.

Our results compare very favorably with those reported in the literature. To solve the XOR problem the following numbers of generations to attain solutions are reported: 90 [33], 513 [34], less than 100 with a minimum solution at generation 200 [35], less than 40 for a neural network with 2 hidden neuron[36]. Note that the values in Table 1 refer to the pruned solutions, larger solutions with two hidden neurons, for example, are obtained much earlier. For the 3 bit parity problem, Yao *et al.* [33] reported an average of 739 generations to get a solution.

The 4-bit parity solution reported by Zhang *et al.* [28] was achieved in fewer generations (9), but a population of 1000 individuals was used, and training by a hillclimbing procedure was performed at each generation. In addition, the resulting network has 6 neurons in the hidden layer and 49 connections, in comparison to the average solution of 5.2 hidden neurons and 26.5 connections obtained with our approach.

In order to evaluate the effect of the size and form of the grid, some experiments were performed with the XOR problem using *node transfer crossover*. The results are shown on Table 5. As can be deduced from the average size of the final solutions in comparison to the initial number of hidden neurons, the pruning strategy consistently reduces the number of hidden neurons. The slightly smaller percentage of solutions found with a small grid indicates that larger grids are better at obtaining solutions, though the average number of generations to achieve the pruned solutions increases. Rows 2 and 3 show the same number of hidden neurons distributed in one and two layers, respectively. Apparently, two layers gives a better performance.

The method was also applied to a 3 output task, where the network must return the sum of two 2 bit input numbers (2 bit adder). This problem was especially hard to solve: only a 2% of the runs were successful. A solution with 5 hidden neurons and 25 connections was obtained. Whitley *et al.* [37] reports a solution with 4 hidden neurons and 29 connections.

Some experiments were also carried out in which both sigmoid and threshold functions were present in the function set. This interesting combination is shown in Figure 7 for the 4 parity and the 2 bit adder problems. This ability of combining different activation functions is a feature offered by very few methods, mostly GP-based.

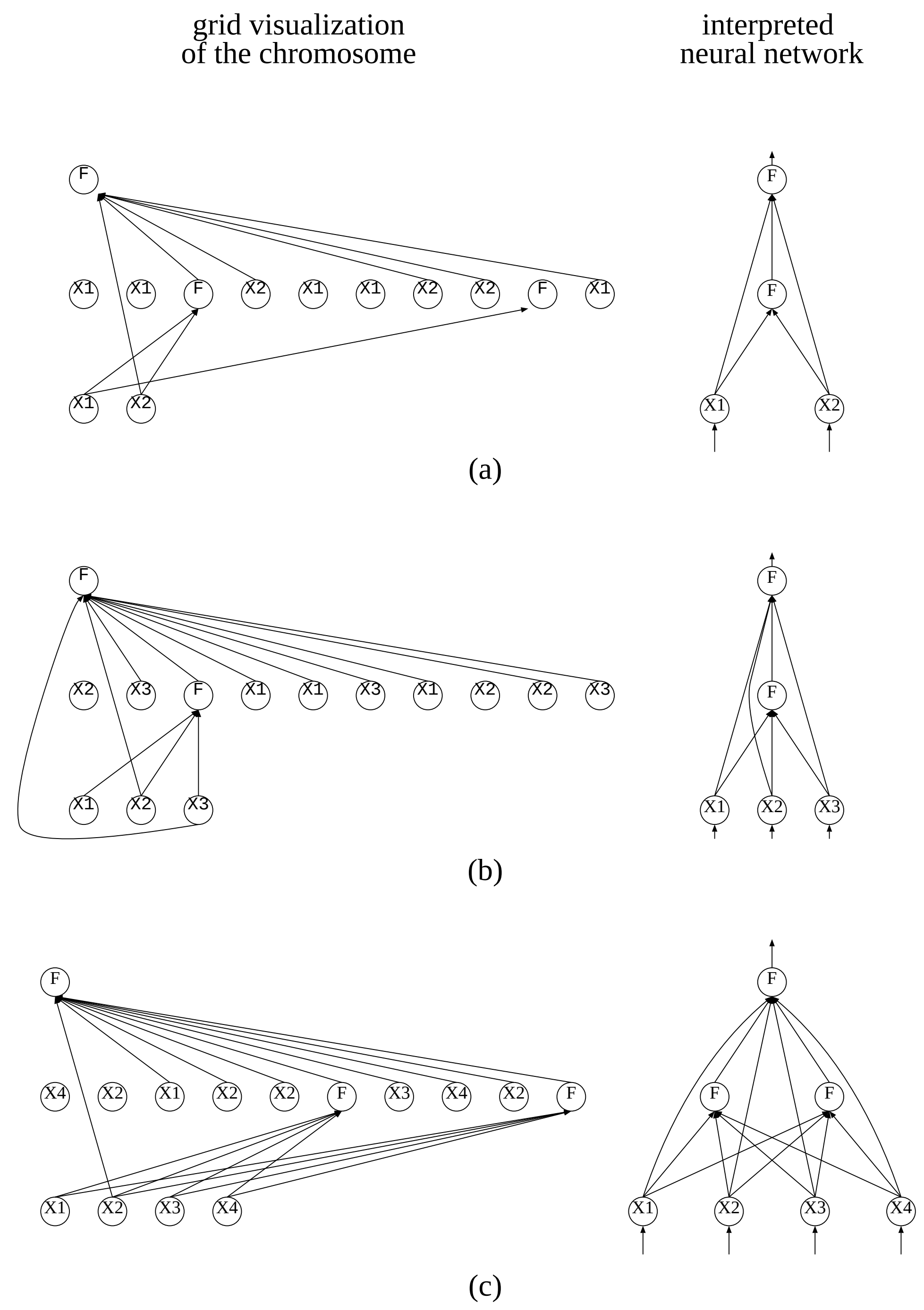
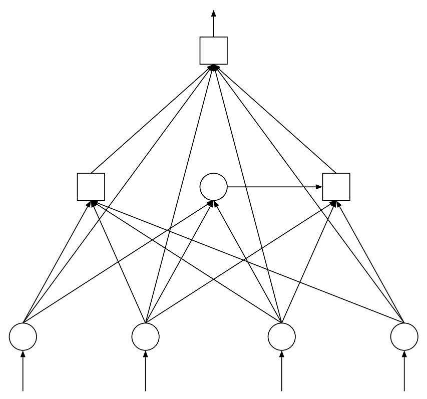


Figure 6: Grid representation and decoded solutions obtained for the XOR (a), 3-odd parity (b) and 4-odd parity (c) problems.



(a)

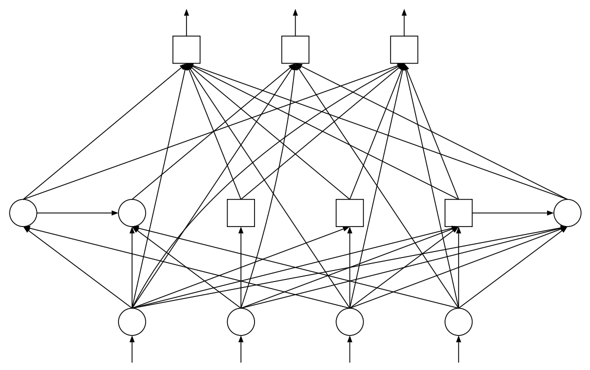


Figure 7: a) A solution for the 4-odd parity problem using a mix of sigmoid and threshold activation functions. b) A solution for the 2 bit adder problem. The squares represent threshold units, while the circles in the hidden layer represent sigmoid units.

task	generations	hidden neurons min/avg/max	connections min/avg/max	effort	solutions found
XOR	66.0	1/1.2/3	5/5.5/11	40,200	100%
3 parity	95.0	1/2.2/9	7/10.7/35	72,000	98%
4 parity	136.3	2/5.2/9	14/26.5/42	720,000	23%

Table 1: Summary of odd parity results using *node transfer crossover* and *node description crossover* with relative probability 3 and 1, respectively.

task	generations	hidden neurons min/avg/max	connections min/avg/max	effort	solutions found
XOR	67.0	1/1.2/4	5/5.5/11	39,800	100%
3 parity	103.1	1/2.0/10	7/10.3/31	78,800	93%
4 parity	121.2	2/5.3/8	14/26.1/37	884,400	19%

Table 2: Summary of odd parity results using *layered sub-graph crossover* and *node description crossover* with relative probability 3 and 1, respectively.

task	generations	hidden neurons min/avg/max	connections min/avg/max	effort	solutions found
XOR	69.5	1/1.5/6	5/6.2/18	40,000	100%
3 parity	105.6	1/2.0/6	7/10.0/24	152,000	75%
4 parity	112.5	5/5.0/5	22/25.0/28	5,472,000	2%

Table 3: Summary of odd parity results using *linear sub-graph crossover* and *node description crossover* with relative probability 3 and 1, respectively.

task	generations	hidden neurons min/avg/max	connections min/avg/max	effort	solutions found
XOR	48.1	1/1.8/8	5/7.0/24	23,800	100%
3 parity	69.3	1/3.3/10	7/13.6/36	141,000	52%
4 parity	130.0	6/6.0/6	29/29.0/29	12,025,800	1%

Table 4: Summary of odd parity results using *one-point crossover* and *node description crossover* with relative probability 3 and 1, respectively.

grid	generations	hidden neurons min/avg/max	connections min/avg/max	effort	solutions found
2x2x1	17.3	1/1.3/2	5/5.6/8	14,000	94%
2x5x1	49.4	1/1.2/4	5/5.6/11	36,000	100%
2x2x3x1	49.7	1/1.4/5	5/5.9/13	32,000	100%
2x4x4x1	56.2	1/1.4/5	5/6.0/16	40,000	100%

Table 5: Results for the XOR problem using different grids.

6 Discussion and further extensions

The results obtained so far are promising, but they were achieved without any optimization of the relative probabilities of the operators. Although some operators seem to perform better than the others, a thorough and systematic investigation of their interaction must be performed, in a broader context involving other tasks.

In the current implementation, the biases have not benefited from the fine tuning accomplished by the *node description crossover*. This may be improved including the biases as connections to a constant node.

With the exception of the Gaussian noise, no other mutation operator has been used in the present study. To keep diversity within the population other mutation operators should be explored.

The extension to recurrent neural networks is a natural one, allowing the method to be applied to a wide range of tasks. Most of the operators implemented do not have to be changed, actually most of them will be simplified.

Finally, the potential of automatic defined functions shall be explored as an alternative to compute the weights of the network. A function to compute the weights can be evolved in parallel to the architecture, using the usual arsenal of functions of genetic programming.

7 Conclusion

In this paper, a new approach to the automatic design of neural networks has been presented. New operators were introduced within the paradigm of genetic programming using a linear encoding together with a graph representation of neural networks.

The representation of the neural network in a linearized form allowed the development of efficient forms of crossover operations and the introduction of a strategy to reduce the complexity of the solutions, whereas the grid description table allowed to control the connectivity properties of the network.

The method was applied to feedforward networks in a variety of binary classification problems showing promising results. Further extensions have been proposed, which will increase the power of the representation and of the operators, allowing it to be applied to an even wider range of practical problems.

8 Acknowledgements

The authors wish to thank the members of the EEBIC (Evolutionary and Emergent Behavior Intelligence and Computation) group for the useful discussions and comments. This research is partially supported by a grant under the British Council-MURST/CRUI agreement.

References

[1] S. Haykin. *Neural networks, a comprehensive foundation.* Macmillan College Publishing Company, Inc., 866 Third Avenue, New York, New York 10022, 1994.

[2] R. Reed. Pruning algorithms: a survey. *IEEE Transactions on Nerual Networks*, 4(5):740–747, 1993.

[3] M. Frean. The upstart algorithm: a method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[4] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532. Morgan Kaufmann, 1990.

[5] D. Chen, C. Giles, G. Sun, H. Chen, Y. Less, and M. Goudreau. Constructive learning of recurrent neural networks. In *IEEE International Conference on Neural Networks (ICNN)*, pages 1196–1201, 1993.

[6] C. Campbell and C. Vicente. The target switch algorithm: a constructive learning procedure for feed-forward neural network. *Neural Computation*, 7:1245–1264, 1995.

[7] X. Yao. An overview of evolutionary computation. *Chinese Journal of Advanced Software Research*, 3(1), 1996. To be published.

[8] D. Fogel. *Evolutionary computation: toward a new philosophy of machine Intelligence.* IEEE PRess, Piscataway, NJ, USA, 1995.

[9] D. Goldberg. *Genetic algorithm in search, optimization and machine learning.* Addison-Wesley, Reading, Massachusets, 1989.

[10] M. Mitchell. *An introduction to genetic algorithms.* MIT Press, Cambridge, Massachusets, USA, 1996.

[11] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Rheingold, New York, NY, 1991.

[12] Michalewicz Zbigniew. *Genetic Algorithms [plus] Data Structures = Evolution Programs.* Springer-Verlag, Berlin, 1994.

[13] S. Harp, T. Samad, and A. Guha. Toward the genetic synthesis of neural networks. In J. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA)*, pages 360–369, San Mateo, CA, USA, 1989.

[14] M. Mandischer. Evolving recurrent neural networks with non-binary encoding. In *Proceedings of the 2nd IEEE Conference on Evolutionary Computation (ICEC)*, volume 2, pages 584–589, Perth, Australia, Nov. 1995.

[15] M. Mandischer. Representation and evolution of neural networks. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms (ANNGA)*, pages 643–649, 1993.

[16] H. Kitano. Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms. *Physica D*, 75:225–238, 1994.

[17] S. Fujita and H. Nishimura. An evolutionary approach to associative memory in recurrent neural networks. *Neural Processing*, 1(2), 1994.

[18] H. Braun and P. Zagorski. ENZO-M - a hybrid approach for optmizing neural networks by evolution and learning. In Y. Davidor, H. Schwefel, and H. Manner, editors, *Parallel Problem Solving from Nature (PPSN3)*, volume 866. Springer-Verlag, 1994. Lecture Notes in Computer Science.

[19] X. Yao and J. Liu. Evolutionary artificial neural networks that learn and generalize well. In *Proceedings of the 1996 IEEE International Conference on Neural Networks*, Washington, DC, Jun. 1996. Submmitted.

[20] V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks*, 5(1):39–53, Jan. 1994.

[21] F. Gruau. *Neural network synthesis using cellular encoding and the genetic algorithm.* PhD thesis, Laboratoire de L'informatique du Parallélisme, Ecole Normale Supériere de Lyon, Lyon, France, 1994.

[22] S. Nolfi and D. Parisi. Growing neural networks. Technical Report PCIA-95-15, Rome, Italy, Jun. 1991.

[23] B. T. Zhang and H. Mühlenbein. Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7(3):199–220, 1993.

[24] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1), 1994.

[25] D. Fogel. Using evolutionary programming to create neural networks that are capable of playing Tic Tac Toe. In *IEEE International Conference on Neural Networks (ICNN).* IEEE Press, 1993.

[26] J. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[27] J. R. Koza. *Genetic Programming, on the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, Massachusets, 1992.

[28] B. Zhang and H. Muehlenbein. Genetic programming of minimal neural nets using Occam's razor. In S. Forrest, editor, *Proceedings of the 5th international conference on genetic algorithms (ICGA'93)*, pages 342–349. Morgan Kaufmann, 1993.

[29] B. Zhang and Muehlenbein. Synthesis of sigma-pi neural networks by the breeder genetic programming. In *Proceedings of IEEE International Conference on Evolutionary Computation (ICEC), World Congress on Computational Intelligence*, pages 318–323, Orlando, Florida, USA, Jun. 1994. IEEE Computer Society Press.

[30] F. Gruau, D. Whitley, and L. Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In J. Koza, D. Goldberg, D. Fogel, and R. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 81–89, Stanford University, CA, USA, Jul. 1996. MIT Press.

[31] R. Poli. Some steps towards a form of parallel distributed genetic programming. In *Proceedings of the First On-line Workshop on Soft Computing*, Aug. 1996.

[32] R. Poli. Discovery of symbolic, neuron-symbolic and neural networks with parallel distributed genetic programming. In *3rd International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)*, 1997.

[33] X. Yao and Y. Shi. A preliminary study on designing artificial neural networks using co-evolution. In *Proceedings of the IEEE Singapore International Conference on Intelligent Control and Instrumentation*, pages 149–154, Jun. 1995.

[34] K. Tang, C. Chan, K. Man, and S. Kwong. Genetic structure for nn topology and weights optimization. In *Porceedings of the International Conference on Genetic Algorithms in Engineering Systems: innovations and applications (GALESIA)*, pages 250–255, Sept. 1995.

[35] D. Dasgupta and D. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In L. Whitley and J. Schaffer, editors, *Proceedings of International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN)*, pages 87–96. IEEE Computer Society Press, Jun. 1992.

[36] D. Fogel, L. Fogel, and V. Porto. Evolving neural networks. *Biological Cybernetics*, 63:487–493, 1990.

[37] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14-3:347–361, 1990.