Evolution of Neural Networks Using Weight Mapping

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
PHONE: (+44)(0121) 414 37 39

Abstract

The application of genetic programming to the evolution of neural networks has been hindered by the inadequacy of parse trees to represent oriented graphs, and by the lack of a good mechanism for encoding the weights. In this work, a hybrid method is introduced, where genetic programming evolves a mapping function to adapt the weights, whereas a genetic algorithm-based approach evolves the architecture. Results on the application of the new method to the evolution of feed-forward and recurrent neural networks are reported.

1 Introduction

The training of artificial neural networks for a particular task can be seen as a mapping of the initial set of random weights into a new set of adapted weights which solves the problem. That means, the process of training defines a function to map the initial set of random weights into the correct ones. For example, the backpropagation training algorithm [1] is an attempt to construct such a *mapping function* iteratively. All training algorithms suffer from a number of problems such as slow convergence, instablility, get trapped in local minima, etc. Genetic programming (GP) has been used to evolve new learning rules [2, 3], which overcome some of these difficulties. These approaches seek to evolve new general purpose training procedures, which are more robust and more efficient than existing ones. Unfortunately, learning procedures always use the same strategy to adapt the weights, disregarding the fact that the error surfaces associated with different tasks present completely different features. Attempts to use genetic programming to evolve the architecture and the weights simultaneously [4, 5] have been marred by the fact that parse trees are not suitable for representing oriented graphs. Alternatively, GP has been used to evolve rules for constructing neural networks [6, 7, 8]. However, this approach imposes constraints on the weights of the neural network.

In this paper, a new approach is discussed which uses genetic programming to automatically build a non-iterative mapping function to adapt the weights. The function is evolved concurrently to the architecture, and is tailored to the task at hand. The architecture and the learning rule are separated in the genotype, and a crossover operator is defined to evolve both parts simultaneously.

The new approach can be described as follows: each individual has its own mapping function to compute the adapted weights from *raw* weights (biases are treated as ordinary weights). The process is illustrated in Figure 1. The individual in Figure 1a has a set of raw weights which are not suitable to solve a particular problem. New values are then computed by applying a mapping function to each raw weight, resulting in the individual in Figure 1b. The mapping function is implemented as the parse tree shown in Figure 1c, to be evolved by genetic programming.

2 Representation

All individuals (genotypes) in the population are structures which have one part to describe the architecture of the encoded network, and a second part to represent the function to map the raw random weights (which are fixed throughout a run) into the values used to evaluate the network performance (see Figure 2). To evolve the architecture, a modified version of the two-dimensional representation introduced in our previous work [9, 10, 11, 12] was used. The structure of the network is encoded as an ordered list of nodes, where each node may be of two types: *terminal* or *neuron*. In the first case, the node is a variable containing an input to the network. In the second case, the node represents a processing element of the encoded network. When the node is a neuron, it is encoded as a list of the incoming connections, represented by indexes, which

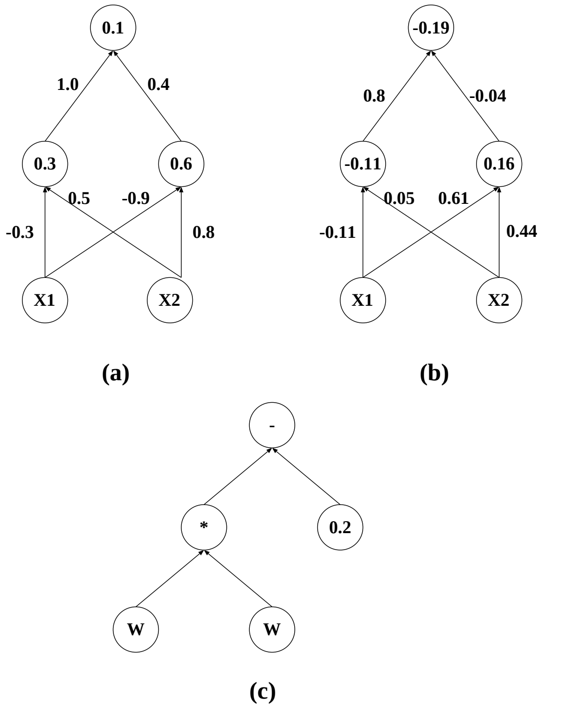


Figure 1: Conversion of raw weights into adapted ones. (a) Network with raw weights (values in the circles are also weights representing biases). (b) Network with adapted weights. (c) Parse tree encoding the mapping function.

indicate the positions (in the list of nodes) of the connected nodes. All individuals in the population have the same number of nodes. (but not of neurons, see below).

To apply the crossover operator (see Section 3), the linear representation just described is interpreted as a two-dimensional arrangement of columns and layers (*grid*). The nodes are mapped onto the grid according to a *description table*, which defines the number of layers and the number of nodes per layer. All individuals in the population use the same table. It is a feature of the population, and is not included in the genotype. The layers and columns of the grid are treated as circular entities, leading to a toroidal grid (the reason for this design will become clear in the description of the crossover operator). For example, by using the description table in Figure 3b, the individual in Figure 3a is interpreted as the two-dimensional representation in Figure 3c, where connections are indicated by links between nodes.

The nodes in the first layer (input layer) are necessarily terminals representing input to the network, whereas the nodes in the last layer (output layer) are necessarily neurons, returning the output of the network. The number of input and output nodes depends on the problem to be tackled. The remaining nodes, called

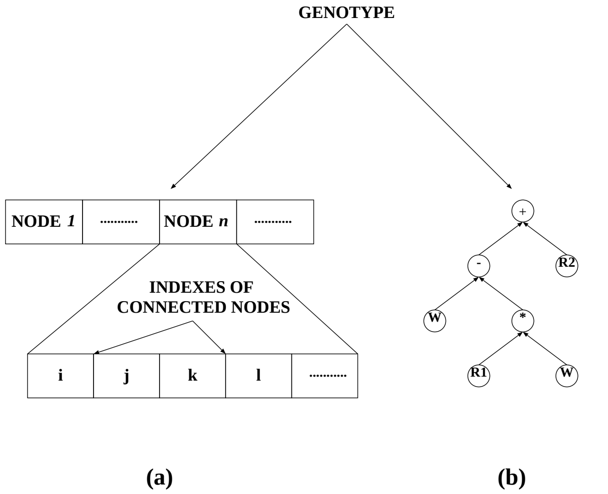


Figure 2: Genotype divided into two parts. (a) First part represents the architecture. (b) Second part is a parse tree to encode the mapping function. *R1* and *R2* are random constants.

internal nodes, constitute the *internal layer(s)*, and they may be either neurons or terminals. As a consequence, networks of different sizes may be represented, since terminals may be present as internal nodes from the beginning, or may be introduced by crossover and mutation. To visualize the actual network encoded in the genotype, connections from terminals in the internal layer can be replaced (and the corresponding terminals removed) with connections from corresponding terminals in the input layer. Subsequently, multiple connections from the same node can be merged into a single one, by adding up their weights (see Figures 3c and d).

The grid representation is very similar to the network, and no decoding procedure is necessary to compute its output. The transformation from genotype to network presented in Figures 3c and 3d is only for visualization purposes. The output of the network can be directly obtained by assigning input values to the terminals, and evaluating the neurons according to their position in the list of nodes.

To assign weights and biases to the architecture, a single ordered list of raw weights is created in the beginning of the evolutionary process, as if all nodes (including those occupied by terminals) were interconnected with the maximum allowed connectivity (feedforward or recurrent). The list also includes raw weights representing biases for all nodes (even for those nodes occasionally occupied by terminals). The set of raw weights is fixed and unique, it is a characteristic of the population. It is used to assign the same raw weights and biases to the connections and neurons of all individuals of the population during fitness evaluation. For each neuron, this is performed by reading the in-

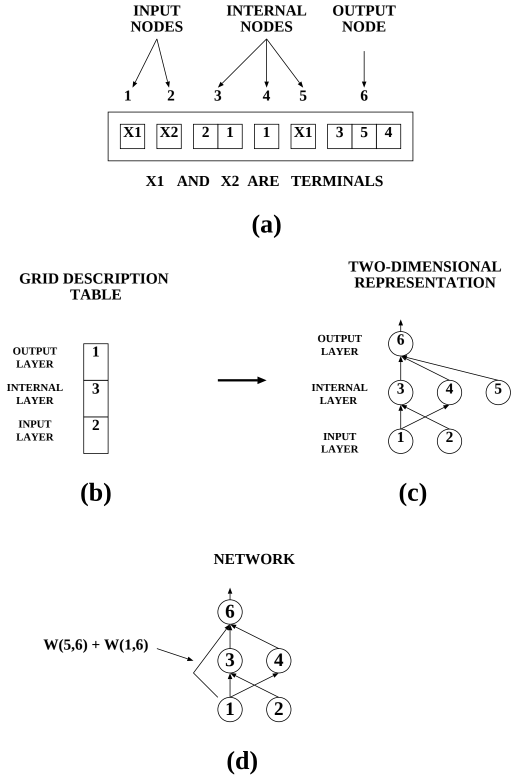


Figure 3: (a) Example of the architecture part of the genotype. Variables *X1* and *X2* are terminals representing input to the network. (b) The table describing the number of layers and the number of nodes per layer of the grid. (c) The two-dimensional representation resulting from the mapping of the nodes in (a), according to the description table in (b). (d) Corresponding network after merging the connections between node 5 and 6, with the connection from node 1 to 6, by adding up their weights and removing node 5.

dexes of the connected nodes, and picking up the corresponding connection weights and biases from the list of raw weights. Afterwards, the mapping function is applied to these raw values, to compute adapted ones.

To define the parse tree in the second part of the genotype, a set of *terminals* (not to be confused with the set of terminals used to define the architecture in the first part of the genotype) and a set of *functions* is defined. The set of terminals includes a variable representing the weight which the function is applied to, and also a variable which is used to initialize random constants when the individuals of the initial population are created (when a parse tree is initially created, the terminals which contain this variable are randomly initialized with real-valued constants). The complexity of the mapping function may depend on the diversity of the adapted weights but, in principle, it is independent of the size of the network.

3 Crossover operator

To evolve both parts of the genotype simultaneously, a combined crossover operator is defined. Firstly, recombination of the architectures of both parents is carried out. Secondly, the parse trees defining the mapping functions of both parents are recombined.

The architectures of the parent genotypes are recombined by selecting a node a in the first parent and a node b in the second parent, and replacing node a with node b in a copy of the first parent (the offspring). Depending on the types of node a and node b, the replacement is carried out as follows:

Both nodes are terminals: This is the simplest case, node b replaces node a, and there is no change either in the topology or in the weights of the network.

Node b is a terminal and node a is a neuron: In this case, node b also replaces node a, but the complexity of the network is reduced, because a neuron is removed from the network.

Node b is a neuron and node a is a terminal: In this situation, the crossover operation increases the complexity of the network, by replacing a terminal with a neuron and increasing the number of hidden neurons in the network. Before node b replaces node a in the offspring, each of its connections is analyzed and possibly modified, depending on whether they are connections from terminals or neurons.

- If the connection is from a neuron, the index of the connected node in the list describing node b is not modified. That means, the connection will still be from the same node after node b replaces node a in the offspring.
- If the connection is from a terminal, the index is modified to point to another node, as if the connection had been rigidly translated from node b to node a. That means, the same horizontal and vertical displacement that exists between nodes a and b is applied to translate horizontally and vertically each incoming connection of node b. For example, in Figure 4, node a is one layer below and one column to the right of node b. Consequently, the connection between node 10 and node b is transformed into a connection between node 7 and node b (see Figures 4b and c). Should the rigid translation of the connection lead to point to an non-existent node outside the limits of the layer or column, the index of the connected node is modified as if the connection had been wrapped around the layer or column. For instance, the connection between node 8 and node b is transformed into

a connection between node 1 and node b (Figures 4b and c).

Here a distinction between recurrent and feedforward connectivity must be made. After the transformation of connections in node b, loops might be created in the offspring. If a feedforward architecture is desired, backward connections are deleted.

This procedure for connection inheritance aims at preserving as much as possible the information present in the connections.

Both nodes are neurons: After modifying node b as in the previous case, node b and node a are combined by selecting two random crossover points, one in each node, and replacing the connections to the right of the crossover point in node a with those to the right of the crossover point in node b, thus creating a new node to replace node a in the offspring. This process can easily create multiple connections between the same two nodes. These connections are deleted before the replacement of node a in the offspring (see Figure 5).

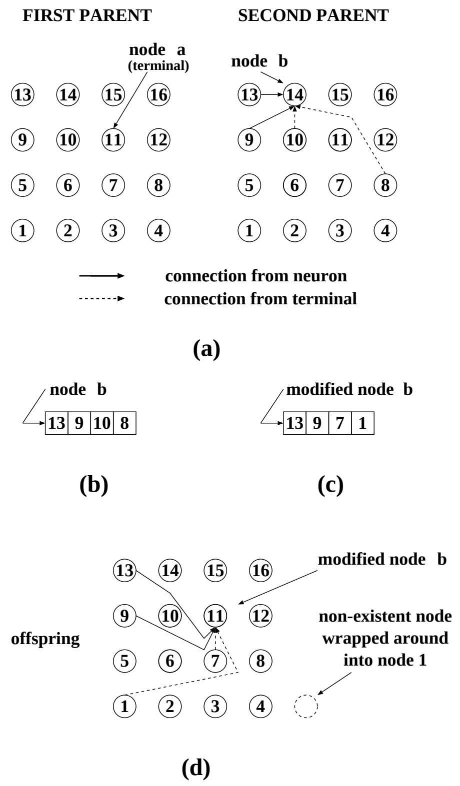


Figure 4: (a) Two-dimensional representation of the parents. (b) Node b. (c) Copy of node b with modified connections. Connections of node b whose indexes indicated connections from terminals received new indexes. (d) Offspring generated by replacing node a with modified node b.

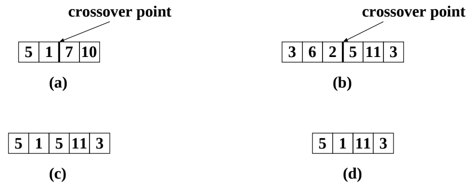


Figure 5: Combination of two neurons. (a) Node a. (b) Modified node b. (c) New node created by crossover with multiple connections. (d) Multiple connections deleted from the new node, to replace node a in the offspring.

Once the recombination of the architectures is complete, a standard genetic programming crossover of the parse trees representing the mapping functions is performed by replacing a random subtree in the second part of the first parent with a random subtree selected from the second part of the second parent. To prevent the excessive growth of the parse trees, a maximum depth is specified, and the selection of the subtrees for swapping is carried out according to this constraint. A bias is also introduced to favor functions in the selection of the roots of the subtrees for crossover [4]. In addition, two terminals representing the variables are never selected as subtrees for crossover, since they encode the same weight.

4 Experimental results

In the experiments described in this section, all individuals were initialized with 10 internal nodes in a single internal layer. Initially, no terminals were present in the internal layer. The raw weights were randomly initialized within the range [-1.0, +1.0]. All individuals in the initial population were initialized with random connectivity. However, it was assured that each node was directly or indirectly connected to the input and output layers. The mean square error of the output of the network for all input patterns was used as fitness function. A threshold activation function was used, $f(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$. Moreover, the following additional conditions were applied to the parse trees:

- Parse trees initialized by the ramped half-and-half procedure [4]. Maximum depth = 8.
- Fraction of random constants included as terminals: 25%. This means that, on the average, when a parse tree was created, the variable representing a weight was assigned to 75% of its terminals, and the remaining 25% of the terminals were randomly initialized with real-valued constants.
- The roots of the subtrees for crossover were selected using a probability distribution which allocated 80% of the crossover points to the internal nodes of the parse trees and 20% to the terminals.

Table 1: Summary of the results on the binary classification problems.

TASK	GEN (σ)	NEURONS				CONNECTIONS				EFFORT
		min	avg	max	σ	min	avg	max	σ	
2 parity (XOR)	4.1 (8.5)	2	8.9	10	1.7	7	32.6	42	7.6	3,000
3 parity	56.0 (74.5)	1	7.8	10	1.8	7	34.6	57	8.6	33,600
4 parity	277.0 (191.7)	3	6.9	10	1.8	21	38.8	57	9.8	224,000
5 parity	406.9 (166.7)	2	4.8	9	2.1	16	31.2	57	11.6	481,600
Symmetry	348.1 (193.8)	5	7.5	10	1.4	24	37.5	48	7.3	248,000

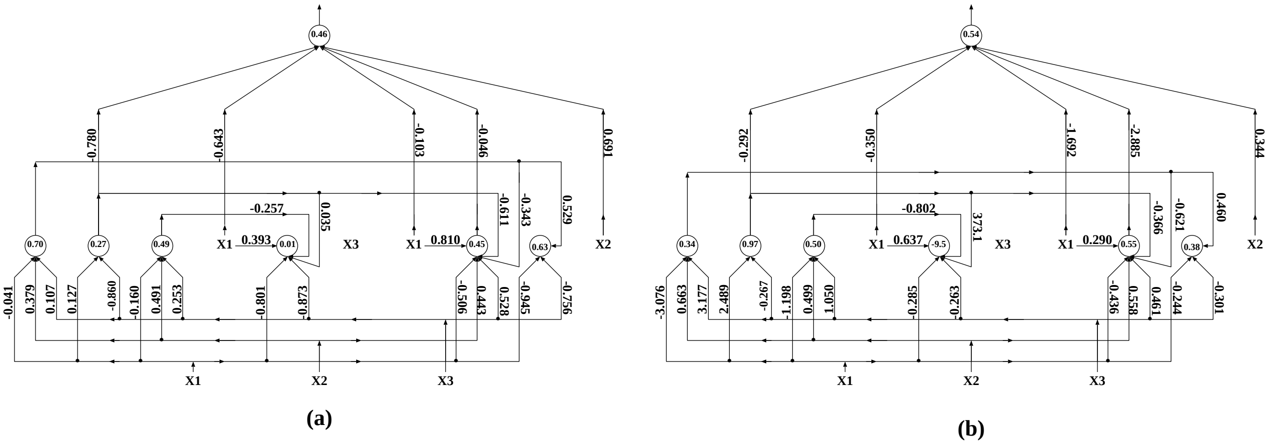


Figure 6: Typical solution to the odd-3 parity problem. (a) With raw weights. (b) With values adapted by the function in Figure 8.

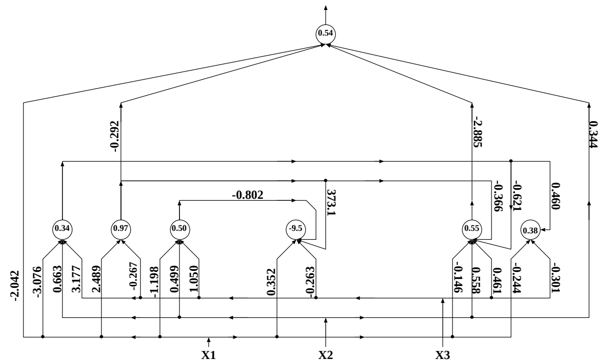


Figure 7: Decoded network for the odd-3 parity problem obtained by replacing connections from terminals in the internal layer (and removing the terminals) with connections from corresponding terminals in the input layer, and merging the resulting multiple connections by adding up their weights.

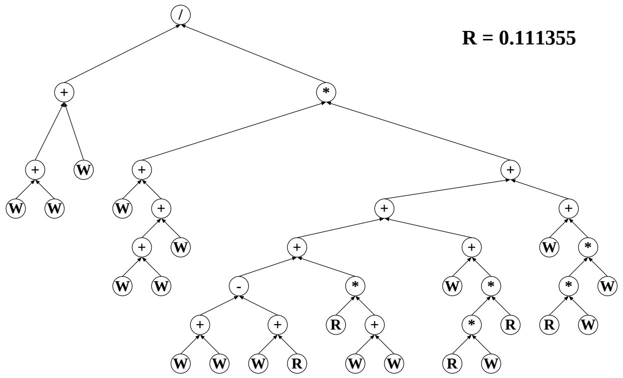


Figure 8: Parse tree representation of the evolved mapping function for the solution to the odd-3 parity problem.

- Function set for the internal nodes of the parse trees: [* , - , + , *PDIV*], where *PDIV* stands for protected division (returns the numerator if the denominator is zero).

4.1 Binary classification problems

The performance of the method proposed was initially tested on a suite of standard benchmark problems present in the literature: the odd-n parity problems, and the 4-symmetry problem (the network is required to classify a 4 bits input bitstring as symmetric around its center or not).

In these experiments, a population of 200 individuals was evolved for a maximum of 500 generations. For each problem, 50 runs were performed with different random seeds. A summary of the results obtained is shown in Table 1. Column 2 represents the average number of generations (standard deviation in brackets). Columns 3, 4, 5 and 6 show the minimum, average, maximum and standard deviation for the number of neurons of the networks evolved, respectively. Columns 7, 8, 9 and 10 give the minimum, average, maximum and standard deviation for the number of connections of the networks evolved, respectively. Column 11 shows the minimal computational effort, i.e. the minimal number of fitness evaluations necessary to obtain a solution with 99% probability in repeated runs [4]. These results compare favourably with those reported in the evolutionary computation literature in terms of the number of generations and the computational effort to find a solution. For example, to solve the XOR problem, the following numbers of generations have been reported: 90 [13], 50 [14], 22 [15]. Koza [4] reports efforts of 80,000, 912,000 and 7,840,000, for the even 3-parity, odd-4 parity, and even-5 parity problems, respectively.

A typical solution to the odd-3 parity problem illustrates the process of building the network. Firstly, the weights are read from the list of raw values and assigned to the architecture evolved. This results in the two-dimensional representation in Figure 6a. Subsequently, the raw weights are adapted by the mapping function in Figure 8 evolved simultaneously to the architecture, leading to the two-dimensional structure in Figure 6b. The two-dimensional representation can then be decoded into the corresponding network shown in Figure 7.

4.2 Tracker problem

To assess the ability of the method to evolve recurrent neural networks, it was applied to the complex problem of finding a control system for an agent whose objective is to track and clear a trail. The particular trail we used, the John Muir trail [16, 17], consists of 89 tiles in a 32x32 toroidal grid.

The tracker starts in the upper left corner, and faces the first position of the trail. The only information available to the tracker is whether the position ahead belongs to the trail or not. Based on this information, at each time step, the tracker can take 4 possible actions: *wait* (doing nothing), *move forward* (one position), *turn right 90º* (without moving) or *turn left 90º* (without moving). When the tracker moves to a position of the trail, that position is immediately cleared. This is a variant of the well known "ant" problem often studied in the GP literature [4].

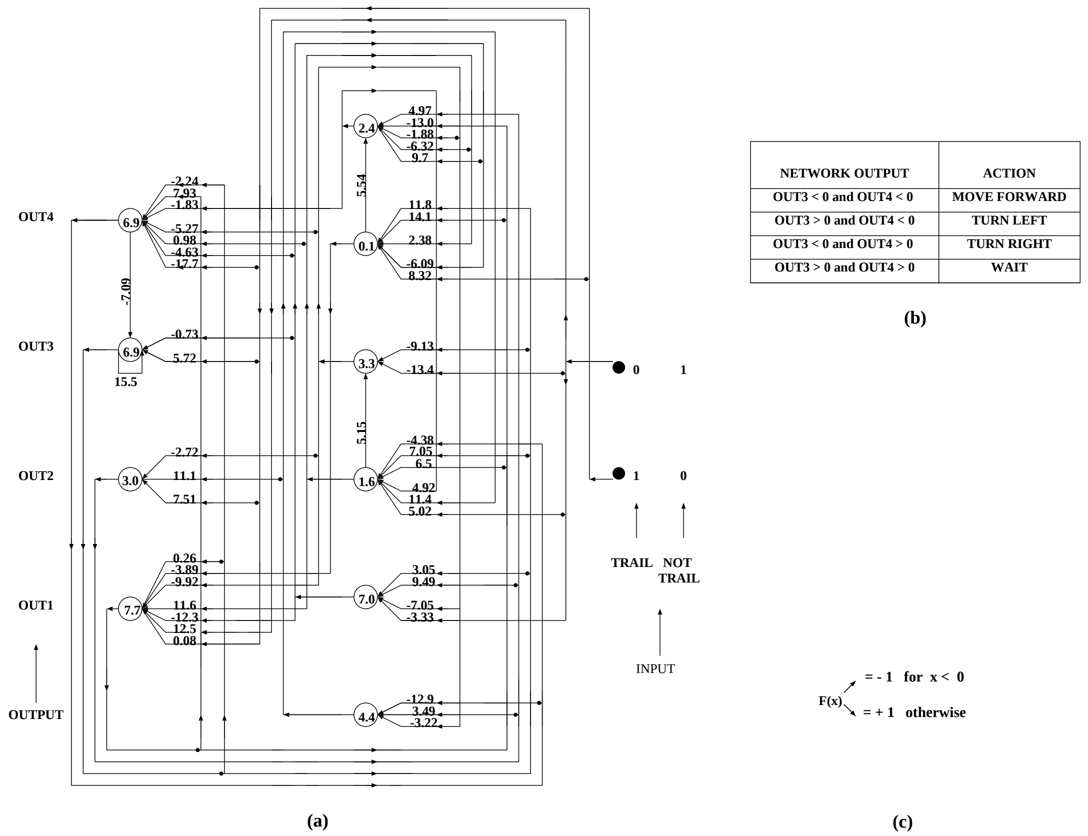
Usually, the information to the tracker is given as a pair of input data [16, 17]: the pair is (1,0) if the position ahead of the current tracker position belongs to the trail, and (0,1) if it does not. The objective is to build a neural network that, at each time step, receives this information, returns the action to be carried out, and clears the maximum number of positions in a specified number of time steps (200 in our experiments). As information about where the tracker is on the trail is not available, it is clear that to solve the problem the neural network must have some sort of memory in order to remember its position. As a consequence, a recurrent network is necessary. Although it might seem to be unnecessary, the *wait* action allows the network to update its internal state while staying at the same position (this can be imagined as "thinking" about what to do next).

This problem is very hard to solve in 200 time steps (in [4], John Koza allotted 400 time steps to follow slightly different trails using GP). However, it is relatively easy to find solutions able to clear up to 90% of the trail [16]. This means that the search space has many local minima which mislead evolution [18].

We used asymmetric recurrent neural networks to solve the problem. A neuron can receive connections from any other neuron (including output neurons). All neurons are evaluated synchronously as a function of the output of the neurons in the previous time step (initially all neurons have null output), and of the current input to the network. A population of 100 individuals was evolved for a maximum of 500 generations. The fitness of an individual was measured by the number of trail positions cleared in 200 time steps.

In 20 independent runs a solution with 6 hidden neurons and 48 connections was found (see Figure 9), amounting to a computational effort of 8,766,000. The solution cleared the entire trail in 199 time steps. It is interesting to note that the solution did not make use of either the *turn left* or the *wait* options to traverse the trail. The average number of positions cleared by the best individuals evolved in the specified maximum number of generations was 80.4 with a standard deviation of 6.6. By assigning additional time steps to the best individuals evolved, in 50% of the runs other networks were also able to clear the trail in less than 290 time steps.

These are very promising results. For comparison, [16] reports a solution with 5 hidden neurons, which clears the trail in 200 time steps. The solution is a handcrafted architecture trained by genetic algorithms using a huge population (65,536 individuals). Using an evolutionary programming approach, [17] reports a network with 9 hidden neurons, evolved in 2,090 generations (population of 100 individuals). The network clears 81 positions in 200 time steps and takes additional 119 ones to clear the entire trail. The authors also report another network evolved in 1,595 generations, which scores 82 positions in 200 time steps.



NETWORK OUTPUT	ACTION
OUT3 < 0 and OUT4 < 0	MOVE FORWARD
OUT3 > 0 and OUT4 < 0	TURN LEFT
OUT3 < 0 and OUT4 > 0	TURN RIGHT
OUT3 > 0 and OUT4 > 0	WAIT

(b)

Figure 9: (a) Solution to the tracker problem. (b) Mapping of the network output into actions. (c) Activation function.

5 Extensions

Although in this work a fixed size representation was used to evolve the structure of the neural network, the use of a weight-mapping function is not limited to this kind of representation. It is possible to define a set of raw weights in variable size representations by considering the number of neurons of the biggest decoded network in the initial population, and the maximum number of connections allowed by connectivity constraints. If in later generations additional weights are necessary (because the networks get bigger), new values can be included in the original set, without disrupting the quality of the mapping function evolved so far.

This opens the possibility of combining the weight mapping approach with any method used to evolve the architecture. For example, grammar-based methods [19, 20] could benefit from this mechanism for encoding the weights. In this case, the genotype would consist of the rules for generating the architecture, and a parse tree to encode the mapping function. This would solve the problem that, due to rewriting mechanisms, grammar-based methods impose a regularity on the distribution of weights in the network. With the weight mapping approach this would not happen. Regularity, if necessary, would be encoded in the mapping function by the evolutionary process.

6 Conclusions

In this paper, a new method for the synthesis of feedforward and recurrent neural networks was presented. We use genetic programming to evolve a mapping function to adapt the weights of neural networks, whereas the architecture is evolved by a genetic algorithm-based method. The method not only has given very promising results, but it also opens new possibilities for genetic programming in the evolution of artificial neural networks.

7 Acknowledgements

The authors wish to thank the members of the EEBIC (Evolutionary and Emergent Behavior Intelligence and Computation) group for useful discussions and comments. This research is partially supported by CNPq and CNEN (Brazil).

References

[1] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In J. McClelland, D. Rumelhart, and the PDP Research Group, editors, *Parallel Distributed Processsing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, Cambridge, Massachusetts, 1986.

[2] S. Bengio, Y. Bengio, and J. Cloutier. Use of genetic programming for the search of a learning rule for neural networks. In *Proceedings of the First Conference on Evolutionary Computation, IEEE World Congress on the Computational Intelligence*, pages 324–327, 1994.

[3] A. Radi and R. Poli. Genetic programming can discover fast and general learning rules for neural networks. In *Proceedings of the Third Annual on Genetic Programming Conference*, pages 314–322, Jul. 1998.

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

[5] E. Vonk, L. Jain, L. Veelenturf, and R. Johnson. Automatic generation of a neural network architecture using evolutionary computation. In *Proceedings of the Elektronic Technology Directions to the Year 2000*, pages 144–149, Adelaide, Australia, May 1995. IEEE Computer Society Press.

[6] D. Whitley, F. Gruau, and L. Pyeatt. Cellular encoding applied to neurocontrol. In *Proceedings of Sixth International Conference on Genetic Algorithms*, pages 460–467. Morgan-kaufmann, 1995.

[7] P. Pratt. Evolving neural networks to control unstable dynamical systems. In A. Sebald and L. Fogel, editors, *Proceedings of the Third Annual Conference on Evolutionary Programming*, pages 191–204, San Diego, USA, Feb. 1994.

[8] C. Friedrich and C. Moraga. Using genetic engineering to find modular structures and activation functions for architectures of artificial neural networks. In *Proceedings of the Fifth Fuzzy Days*, pages 150–161, 1997.

[9] J. Pujol and R. Poli. Evolving the topology and the weights of neural networks using a dual representation. *Special Issue on Evolutionary Learning of the Applied Intelligence Journal*, 8(1):73–84, Jan. 1998.

[10] J. Pujol and R. Poli. Efficient evolution of asymmetric recurrent neural networks using a pdgp-inspired two-dimensional representation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming (EUROGP)*, volume 1391 of *Lecture Notes in Computer Science*, pages 130–141. Springer-Verlag, 1998.

[11] J. Pujol and R. Poli. Evolving neural networks using a dual representation with a combined crossover operator. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, pages 416–421, 1998.

[12] J. Pujol and R. Poli. Dual network representation applied to the evolution of neural controllers. In V. Porto, N. Saravanan, D. Waagen, and A. Eiben, editors, *Proceedings of the Seventh Annual Conference on Evolutionary Programming (EP)*, volume 1447 of *Lecture Notes in Computer Science*, pages 637–647. Springer-Verlag, 1998.

[13] 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.

[14] T. Nagao, T. Agui, and H. Nagahashi. Structural evolution of neural networks having arbitrary connections by a genetic method. *IEICE Transactions on Information and Systems*, E76D(6):689–697, Jun. 1993.

[15] M. Sase, K. Matsui, and Y. Kosugi. Intergenerational architecture adaptation of neural networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, volume 3, pages 2941–2944, Nagoya, Japan, Oct. 1993.

[16] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life: The genesys/tracker system. In C. Langton, C. Taylor, J. Farmer, and S. Rasmussen, editors, *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*, volume X, pages 549–578. Addison-Wesley, 1991.

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

[18] W. Langdon and R. Poli. Why ants are hard. Technical report CSRP-98-04, School of Computer Science, The University of Birmingham, 1998.

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

[20] 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.