

Efficient Evolution of Asymmetric Recurrent Neural Networks Using a PDGP-inspired Two-dimensional 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

Abstract. Recurrent neural networks are particularly useful for processing time sequences and simulating dynamical systems. However, methods for building recurrent architectures have been hindered by the fact that available training algorithms are considerably more complex than those for feedforward networks. In this paper, we present a new method to build recurrent neural networks based on evolutionary computation, which combines a linear chromosome with a two-dimensional representation inspired by Parallel Distributed Genetic Programming (a form of genetic programming for the evolution of graph-like programs) to evolve the architecture and the weights simultaneously. Our method can evolve general asymmetric recurrent architectures as well as specialized recurrent architectures. This paper describes the method and reports on results of its application.

1 Introduction

The ability to store temporal information makes recurrent neural networks (RNNs) ideal for time sequence processing and dynamical systems simulation. However, building RNNs is far more difficult than building feedforward neural networks. Constructive and destructive algorithms, which combine training with structural modifications which change the complexity of the network, have been proposed for the design of recurrent networks [1, 2], but their application has been hindered by the fact that training algorithms for recurrent architectures are considerably more complex than their feedforward counterparts [3, 4, 5, 6, 7].

Recently, new promising approaches based on evolutionary algorithms, such as evolutionary programming (EP) [8] and genetic algorithms (GAs) [9], have been applied to the development of artificial neural networks (ANNs). Approaches based on EP operate on the neural network directly, and rely exclusively on mutation [10, 11, 12, 13] or combine mutation with training [14]. Methods based on genetic algorithms usually represent the structure and the weights of ANNs as a string of bits or as a combination of bits, integers and real numbers [15, 16, 17, 18, 19, 20], and perform the crossover operation as if the network were a linear structure. However, neural networks cannot naturally be represented as vectors. They are oriented graphs, whose nodes are neurons and whose arcs are synaptic connections. Therefore, it is arguable that any efficient approach to evolve ANNs should use operators based on this structure.

Some recent work based on genetic programming (GP) [21], originally developed to evolve computer programs, is a first step in this direction. For example, in [21, 22] neural networks have been represented as parse trees which are recombined using a crossover operator which swaps subtrees representing subnetworks. However, the graph-like structure of neural networks is not ideally represented directly with parse trees either. Indeed, an alternative approach based on GP, known as cellular encoding [23], has recognized this and used an indirect network representation in which parse trees represent rules which grow a complete network from a tiny neural embryo. Although very compact, this representation enforces a considerable bias on the network architectures that can be achieved.

Recently, a new form of GP, Parallel Distributed Genetic Programming (PDGP) [24], in which programs are represented as graphs instead of parse trees, has been applied to the evolution of neural networks [25]. The method allows the use of more than one activation function in the neural network but it does not include any operators specialized in handling the connection weights. Nonetheless, the results were encouraging and led us to believe that a PDGP-inspired representation, with specialized operators acting on meaningful building blocks, could be used to efficiently evolve neural networks. Indeed, in [26] we improved and specialized PDGP by introducing a dual representation, where a linear description of the network was dynamically associated to a two-dimensional grid. Several specialized genetic operators, some using the linear description, others using the grid, allowed the evolution of the topology and the weights of moderately complex neural networks very efficiently. In [27, 28], we proposed a combined crossover operator which allowed the evolution of the architecture, the weights and the activation function of feedforward neural networks concurrently.

In this paper, we extend the combined crossover operator to the design of recurrent networks. In the following sections, our representation and operators are described, and we report on results of the application of the paradigm to the relatively complex task of tracking and clearing a trail.

2 Representation

In PDGP, instead of the usual parse tree representation used in GP, a graph representation is used, where the functions and terminals are allocated in a two-dimensional grid of fixed size and shape. The grid is particularly useful to solve problems whose solutions are graphs with a natural layered structure, like neural networks.

However, this representation can make inefficient use of the available memory if the grid has the same shape for all individuals in the population. In fact, in this case there is no need to represent the grid explicitly in each individual. Also, in some cases, it is important to be able to abstract from the physical arrangement of neurons into layers and to only consider the topological properties of the net. To do this, it is more natural to use a linear genotype.

These limitations led us to propose [26] a dual representation, in which a linear chromosome is converted, when needed, into the grid-based PDGP representation. The dual representation includes every detail necessary to build and use the network: connectivity, weights and the activation function of each neuron. A chromosome is an

ordered list of nodes (see Figure 1). An index indicates the position of the nodes in the chromosome. All chromosomes have the same number of nodes.

Like in standard genetic programming, the nodes are of two kinds: functions and terminals. The functions represent the neurons of the neural network, and the terminals are the variables containing the input to the network (see Figure 2a).

When the node is a neuron, it includes the activation function and the bias, as well as the indexes of other nodes sending signals to the neuron and the weights necessary for the computation of its output. Multiple connections from the same node are allowed.

When necessary (see Section 3), the linear representation just described is transformed into the two-dimensional representation used in PDGP. A description table defines the number of layers and the number of nodes per layer of the two-dimensional representation (see Figure 2b and c). The nodes of the linear chromosome are mapped onto this representation according to the description table. The connections of the network are indicated by links between nodes in the two-dimensional representation. The description table is a characteristic of the population and it is not included in the genotype. The two-dimensional representation may have any number of layers, each of different size. This feature may be used to constrain the geometry of the network (e.g. to obtain encoder/decoder nets) and can also be used to guide specialized crossover operators [26, 27].

The nodes in the first layer are terminals, whereas the nodes in the last layer represent the output neurons of the network. The number of input and output nodes depends on the problem to be solved. The remaining nodes, called *internal nodes*, constitute the internal layer(s), and they may be either neurons or terminals.

Although the size of the chromosome is fixed for the entire population, the neural networks represented may have different sizes. This happens because terminals may be present as internal nodes from the beginning, or may be introduced by crossover and mutation (this is discussed in Section 3). They are removed from the network during the decoding phase, which is performed before each individual is evaluated. Connections from the removed terminals are replaced with connections from corresponding terminals in the input layer (see Figures 2c and d).

Our model allows the use of more than one activation function, so that a suitable combination of activation functions can be evolved to solve a particular problem (this is discussed in the next section).

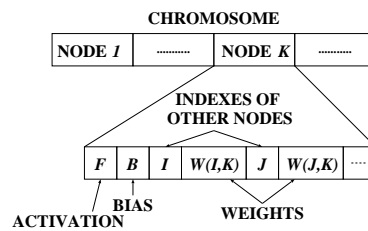


Fig. 1. Chromosome and node description.

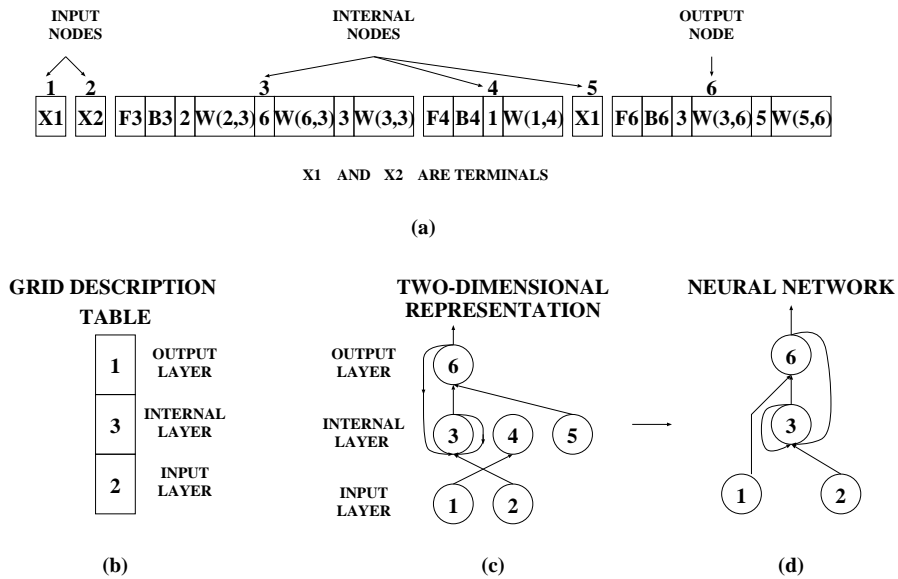


Fig. 2. (a) Example of a chromosome. (b) The table describing the number of nodes per layer of the network. (c) The two-dimensional representation resulting from the mapping of the nodes of the chromosome in (a), according to the description table in (b) (note the intron in the internal layer). (d) Resulting neural network after elimination of introns and transfer of connection from terminals in the first layer (Note that node 5 is the same terminal as node 1).

3 Crossover

By experimenting with different combinations of crossover and mutation operators in our previous work [26], we have drawn the conclusion that, for the evolution of neural networks, it is important to use operators which induce a fitness landscape as smooth as possible, and that it is also important to treat connections from terminals differently from connections from functions (neurons).

The crossover operator proposed in this paper works by randomly selecting a node a in the first parent and a node b in the second parent (see Figure 3a), and by replacing node a in a copy of the first parent (the offspring). Depending on the types of node a and node b , the replacement of node a 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 function 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 function and node a is a terminal In this situation, the cross-over operation increases the complexity of the network. A temporary node, c , is created as a copy of node b . Before node c replaces node a in the offspring, each of its connections is analyzed and possibly modified, depending on whether they are connections from terminals or functions.

- If the connection is from a function, the index of the connected node is not modified.
- 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 . For example, the connection between node 10 and node b in Figures 3a and b is transformed into a connection between node 7 and node c in Figures 3c and d. In some cases, translating the connection rigidly might lead to point to a non-existent node outside the limits of the layer. In this case, the index of the connected node is modified as if the connection had been wrapped around the layer. For instance, the connection between node 8 and node b in Figures 3a and b is transformed into a connection between node 1 and node c in Figures 3c and d. If the rigid translation of the connection requires the presence of a node below the input layer or above the output layer, it is wrapped around from top to bottom (or vice-versa), as if the columns were circular.

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

Both nodes are functions This is the most important case. By combining the description of two functions, the topology and the weights of the network can be changed. After creating node c as described above, its description and the description of node a are combined by selecting two random crossover points, one in each node, and by replacing the connections to the right of the crossover point in node a with those to the right of the crossover point in node c , thus creating a new node to replace node a in the offspring. See Figure 4.

This process can easily create multiple connections between the same two nodes. These are very important because their net effect is a fine tuning of the connection strength between two nodes. However, as this may reduce the efficiency of the search, we only allow a prefixed maximum number of multiple connections. If more than the allowed maximum number of multiple connections are created, some of them are deleted before the replacement of node a in the offspring.

Modification of the activation function and bias of a node is not performed with our crossover operator. However, this can be indirectly accomplished by replacing a function with a terminal, which can later be replaced with a function with different features.

This crossover operator can not only evolve the topology and strength of the connections in a network, but also the number of neurons and the neurons themselves, by replacing their activation functions and biases. Similarly to GP, even if both parents are equal, the offspring may be different. This helps to keep diversity.

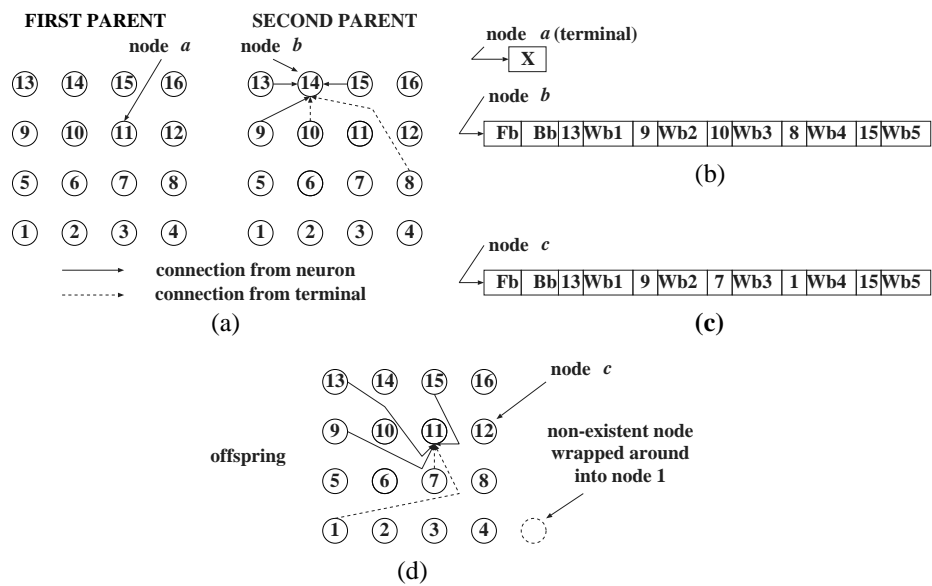


Fig. 3. (a) Two-dimensional representation of the parents. For clarity, only connections relevant to the operation are shown. (b) Nodes *a* and *b*. (c) Node *c* is a copy of node *b* with modified connections. The connections of node *b* whose indexes indicated connections from terminals received new indexes. (d) Offspring generated by replacing node *a* with node *c*.

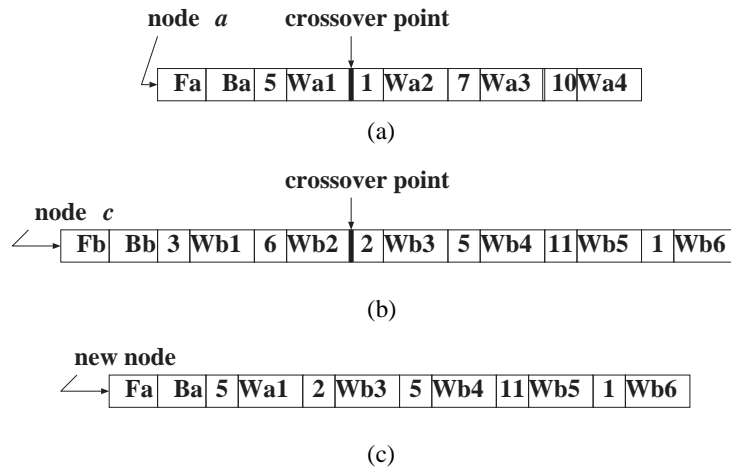


Fig. 4. Combination of two functions. (a) Node *a*. (b) Node *c*. (c) New node created to replace node *a* in the offspring. Note the multiple connections created.

4 Mutation

Our dual representation 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 the representation, as the size of the chromosomes is constant. However, a hidden neuron may be replaced with a terminal or vice-versa, and this may be used to reduce or increase the complexity of the network, within predefined limits.

5 Application domain

Our method can deal with fully or partially connected networks. Moreover, specialized architectures with constrained connectivity can be also evolved by initializing the population with individuals of proper connectivity. Undesirable connections possibly created in the offspring are ignored when the genotype is decoded into the phenotype before fitness evaluation is carried out. These connections are not deleted from the genotype, they are introns not expressed in the phenotype, and may become active again later.

For example, in Elman networks [29] the context units can be represented by feedback connections from the hidden neurons to themselves. The remaining connections in the network are feedforward connections. If connections not complying with these constraints are created, they can be ignored during network evaluation. The same applies to cascade correlation recurrent networks [1], where only feedback connections from a hidden neuron to itself are allowed. In NARMA and NARX networks [30, 31, 32] the feedback from the output neuron to the hidden neurons through the delayed input units can be automatically represented in our method by including additional terminals to input this information to the network.

This flexibility to evolve architectures of quite different connectivities makes it possible for our method to cover a wide range of tasks.

6 Tracker problem

To assess the performance of the method proposed, it was applied to the moderately 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 [33, 11], consists of 89 tiles in a 32x32 grid (black squares in Figure 5). The grid is considered to be toroidal.

The tracker starts in the upper left corner, and faces the first position of the trail (see Figure 5). 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 90°* (without moving) or *turn left 90°* (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 [21].

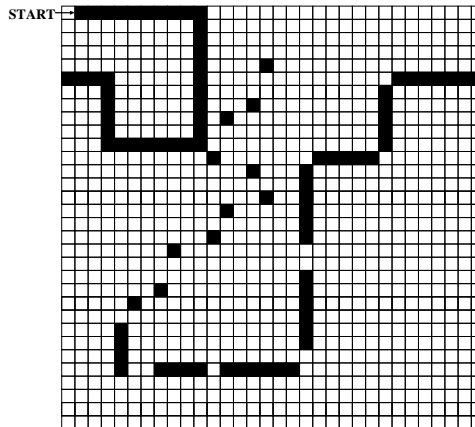


Fig. 5. John Muir trail.

Usually, the information to the tracker is given as a pair of input data [33, 11]: 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 [21], 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 (in [33] Jefferson et al. present a statistical evaluation of this subject using finite automata and neural networks). This means that the search space has many local minima which mislead evolution, as confirmed by Langdon & Poli [34], who studied the fitness landscape of the ant problem for GP. This makes the problem a good benchmark to test the ability of our method to evolve recurrent neural networks.

7 Experimental Results

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, and of the current input to the network. Initially, all neurons have null output.

In all experiments a population of 100 individuals was evolved. All individuals were initialized with 10 internal nodes in a 2-10-2 grid. The weights and biases were randomly

initialized within the range $[-1.0, +1.0]$. We used crossover with a randomly created individual as mutation operator and a threshold activation function. A maximum of 5 multiple connections was allowed between each pair of nodes. We used a generational genetic algorithm with tournament selection (tournament size = 4). The crossover and mutation probabilities were 70% and 5%, respectively. The fitness of an individual was measured by the number of trail positions cleared in 200 time steps. The population was allowed to evolve for a maximum of 500 generations.

In 20 independent runs, the average number of positions cleared by the best individuals was 81.5 (standard deviation of 6.4). In spite of this, we found a network with 7 hidden neurons, which cleared the trail in 199 time steps (see Figure 6). Moreover, by assigning additional time steps to the best individuals evolved, other networks were able to clear the trail. As a result, in 60% of the runs, the best individuals evolved cleared the 89 positions in less than 286 time steps, and 50% in less than 248 time steps. An unexpected feature of the solution we found is that, although available, it does not make use of either the *turn left* or the *wait* options to move along the trail.

These are very promising results. For comparison, Jefferson et al. [35] report a solution with 5 hidden neurons, which clears the trail in 200 time steps. The solution is a hand-crafted architecture trained by a genetic algorithm using a huge population (65,536 individuals). Using an evolutionary programming approach, Angeline et al. [11] report a network with 9 hidden neurons, evolved in 2090 generations (population of 100 individuals). The network clears 81 positions in 200 time steps and takes additional 119 time steps to clear the entire trail. They also report another network evolved in 1595 generations, which scores 82 positions in 200 time steps.

In order to show the ability of our method to evolve neural networks with more than one activation function, 20 additional independent runs were carried out with two activation functions: the threshold activation function of the previous experiment and the hyperbolic tangent. We found a network with 6 hidden neurons and 56 connections able to clear the trail in 259 steps.

8 Conclusion

In this paper, a new approach to the automatic design of recurrent neural networks has been presented, which makes natural use of their graph structure. The approach is based on a dual representation and a new crossover operator. The method was applied to evolve asymmetric recurrent neural networks with promising results.

In future research, we intend to extend the power of the method, and to apply it to a wider range of practical problems.

9 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 a grant under the British Council-MURST/CRUI agreement and by CNPq (Brazil).

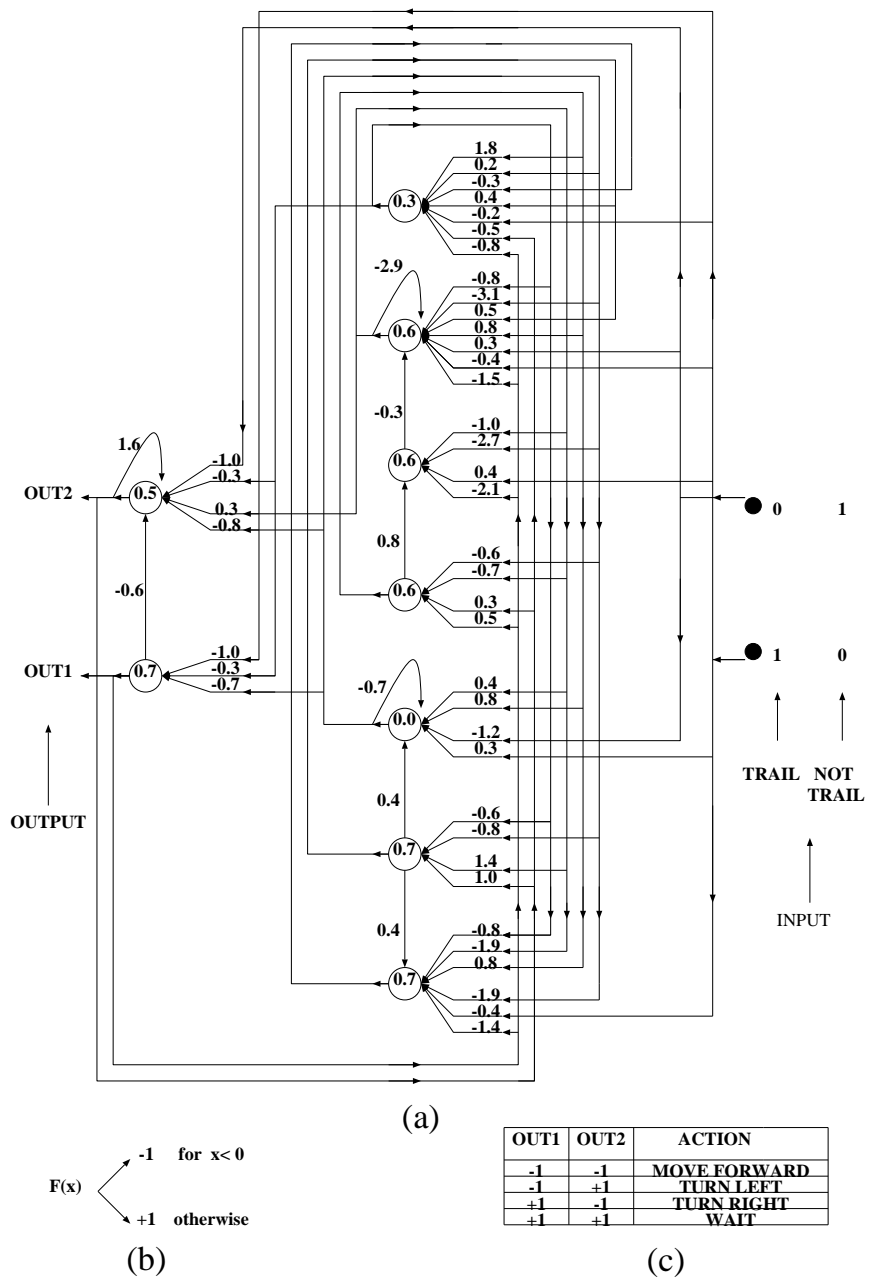


Fig. 6. Full solution to the tracker problem. (a) Neural network. Values in the circles are the biases. (b) Activation function used. (c) Mapping of the network output into actions.

References

1. S. E. Fahlman and C. Lebiere. A recurrent cascade-correlation learning architecture. In R. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 190–196. Morgan Kaufmann, 1991.
2. C. L. Giles and W. Omlin. Pruning recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(5):848–855, 1994.
3. S. Haykin. *Neural networks, a comprehensive foundation*. Macmillan College Publishing Company, Inc., 866 Third Avenue, New York, New York 10022, 1994.
4. J. Hertz, K. Anders, and R. G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Redwood, California, 1991.
5. F. J. PINEDA. Generalization of backpropagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, Nov. 1987.
6. D. R. Hush and B. G. Horne. Progress in supervised neural networks. *IEEE Signal Processing Magazine*, pages 8–39, Jan. 1993.
7. A. M. Logar, E. M. Corwin, and W. J. B. Oldham. A comparison of recurrent neural network learning algorithms. In *IEEE International Conference on Neural Networks*, pages 1129–1134, Stanford University, 1993.
8. T. Bäck, G. Rudolph, and H. Schwefel. Evolutionary programming and evolution strategies: Similarities and differences. In *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 11–22. Evolutionary Programming Society, 1993.
9. D. Goldberg. *Genetic algorithm in search, optimization and machine learning*. Addison-Wesley, Reading, Massachusetts, 1989.
10. K. Lindgren, A. Nilsson, M. Nordahl, and I. Rade. Evolving recurrent neural networks. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms (ANNGA)*, pages 55–62, 1993.
11. 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.
12. J. McDonnell and D. Waagen. Neural network structure design by evolutionary programming. In D. Fogel and W. Atmar, editors, *Proceedings of the Sec. Annual Conference on Evolutionary Programming*, pages 79–89, La Jolla, CA, USA, Feb. 1993. Evolutionary Programming Society.
13. J. McDonnell and D. Waagen. Evolving recurrent perceptrons for time-series modelling. *IEEE Transactions on Neural Networks*, 5(1):24–38, Jan. 1994.
14. X. Yao and Y. Liu. Evolving artificial neural networks through evolutionary programming. In *Proceedings of the 5th Annual Conference on Evolutionary Programming*, San Diego, CA, USA, Feb/Mar 1996. MIT Press.
15. V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks*, 5(1):39–53, 1994.
16. D. Whitley, S. Dominic, R. Das, and C. Anderson. Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284, 1993.
17. 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.
18. J. Santos and R. Duro. Evolutionary generation and training of recurrent artificial neural networks. In *Proceedings of the first IEEE Conference on Evolutionary Computation (ICEC)*, volume 2, pages 759–763, Orlando, FL, USA, Jun. 1994.
19. S. Bornholdt and D. Graudenz. General asymmetric neural networks and structure design by genetic algorithms. *Neural Networks*, 5:327–334, 1992.

20. F. Marin and F. Sandoval. Genetic synthesis of discrete-time recurrent neural network. In *Proceedings of International Workshop on Artificial Neural Network (IWANN)*, pages 179–184. Springer-Verlag, 1993.
21. J. R. Koza. *Genetic Programming, on the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.
22. 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.
23. F. Gruau. *Neural network synthesis using cellular encoding and the genetic algorithm*. PhD thesis, Laboratoire de L'informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Lyon, France, 1994.
24. R. Poli. Some steps towards a form of parallel distributed genetic programming. In *Proceedings of the First On-line Workshop on Soft Computing*, pages 290–295, Aug. 1996.
25. 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.
26. J. C. F. Pujol and R. Poli. Evolution of the topology and the weights of neural networks using genetic programming with a dual representation. Technical report CSRP-97-07, The University of Birmingham, School of Computer Science, 1997.
27. J. C. F. Pujol and R. Poli. A new combined crossover operator to evolve the topology and the weights of neural networks using a dual representation. Technical report CSRP-97-12, The University of Birmingham, School of Computer Science, 1997.
28. J. C. F. Pujol and R. Poli. Evolving neural controllers using a dual network representation. Technical report CSRP-97-25, The University of Birmingham, School of Computer Science, 1997.
29. J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.
30. J. T. Connors and R. D. Martin. Recurrent neural networks and robust series prediction. *IEEE Transactions on Neural Networks*, 5(2):240–254, Mar. 1994.
31. T. Lin, B. G. Horne, P. Tino, and C. L. Giles. Learning long-term dependencies in narx recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6), Nov. 1996.
32. H. T. Siegelmann, B. G. Horne, and C. L. Giles. Computational capabilities of recurrent narx neural networks. *IEEE Transactions on Systems, Man and Cybernetics*, 27(8), Mar. 1997.
33. R. Collins and D. Jefferson. Antfarm: toward simulated evolution. 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. Addison-Wesley, 1991.
34. W. Langdon and R. Poli. Why ants are hard. Technical report CSRP-98-04, The University of Birmingham, School of Computer Science, 1998.
35. 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. Addison-Wesley, 1991.