# On the Design of an Artificial Life Simulator

Dara Curran, Colm O'Riordan

Dept. of Information Technology
National University of Ireland, Galway.

**keywords:** Artificial life, simulation, neural networks, genetic algorithms

**Abstract.** This paper describes the design of an artificial life simulator. The simulator uses a genetic algorithm to evolve a population of neural networks to solve a presented set of problems. The simulator has been designed to facilitate experimentation in combining different forms of learning (evolutionary algorithms and neural networks). We present results obtained in simulations designed to examine the effect of individual life–time learning on the population's performance as a whole.

## 1 Introduction

Genetic algorithms have long been used as an efficient approach to solution optimisation[1]. Based on a Darwinian evolutionary scheme, potential solutions are mapped to genetic codes which, in the case of the canonical genetic algorithm, are represented by bit patterns. Each of these solutions is tested for validity and a portion are selected to be combined to create the next generation of solutions. Using this mechanism in a iterative manner, the approach has been shown to solve a variety of problems[2].

Artificial neural networks are a method of machine learning based on the biological structure of the nervous systems of living organisms. A neural network is composed of nodes and interconnecting links with associated adjustable weights which are modified to alter a network's response to outside stimuli. Through a process of training, a neural network's error can be iteratively reduced to improve the network's accuracy.

The focus of this paper is on the design and development of an artificial life simulator which combines both genetic algorithm and neural network techniques. An initial set of experiments is also presented, which examines the relationship between life–time learning and increased population fitness.

The next section discusses in more detail some limitations of genetic algorithms and neural networks when used in isolation and presents some of the successful work which has been carried out using a combination of the two approaches. Section 3 presents the simulator's architecture, including the encoding mechanism employed and Section 4 outlines the initial experiments carried out with the simulator.

## 2 Related Work

While the individual use of genetic algorithms and neural networks has been shown to be successful in the past, there are disadvantages associated with both approaches. The learning algorithm employed by neural network implementations is frequently based on back propagation or a similar gradient descent technique employed to alter weighting values. These algorithms are liable to become trapped in local maxima and in addition, it is difficult to foresee a neural network architecture design to solve a given problem[3, 4].

Genetic algorithms may also become trapped in local maxima and, furthermore, they require that potential solutions to a given problem be mapped to genetic codes. Not all problems are readily converted to such a scheme[1].

The combination of neural networks and genetic algorithms originally stemmed from the desire to generate neural network architectures in an automated fashion using genetic algorithms[5, 6]. The advantage of this approach is that neural networks can be selected according to a variety of criteria such as number of nodes, links and overall accuracy. The neural network component, on the other hand, provides computational functionality at an individual level within the genetic algorithm's population. The combination of genetic algorithms and neural networks has since been proven successful in a variety of problem domains ranging from the study of language evolution[7] to games[8].

## 3 Simulator Architecture

The architecture of the simulator is based on a hierarchical model (figure 1). Data propagates from the simulator's interface down to the simulator's lowest level. The neural network and genetic algorithm layers generate results which are then fed back up to the simulator's highest level.

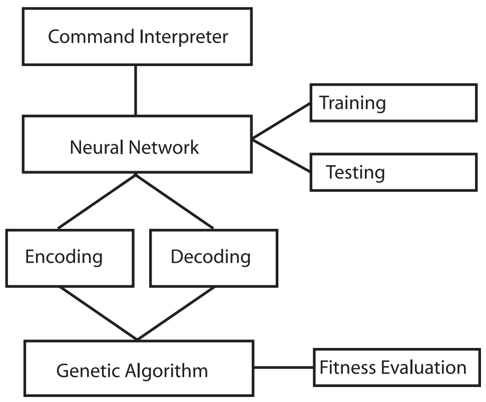

**Fig. 1.** *Simulator Architecture*

### 3.1 Command Interpreter Layer

The command interpreter is used to receive input from users in order to set all variables used by the simulator. The interpreter supports the use of scripts, allowing users to store parameter information for any number of experiments.

### 3.2 Neural Network Layer

The neural network layer generates a number of networks using variables set by the command interpreter and initialises these in a random fashion. Once initialisation is complete, the network layer contains a number of neural networks ready to be trained or tested.

**Training** Several algorithms exist to alter the network's response to input so as to arrive at the correct output. In this system, the back propagation algorithm is used. Error reduction is achieved by altering the value of the weights associated with links connecting the nodes of the network. Each exposure to input and subsequent weight altering is known as a training cycle.

**Testing** Testing allows the simulator to ascertain how well each network solves a given problem. The output for each network is used by the selection process in the genetic algorithm layer.

### 3.3 Encoding and Decoding Layers

To perform genetic algorithm tasks, the neural network structures must be converted into gene codes on which the genetic algorithm will perform its operations. This conversion is carried out by the encoding layer. Once the genetic algorithm has generated the next generation, the decoding layer converts each gene code back to a neural network structure to be trained and tested. The encoding and decoding layers follow the scheme outlined in Section 3.5.

### 3.4 Genetic Algorithm Layer

The genetic algorithm layers is responsible for the creation of successive generations and employs three operators: selection, crossover and mutation.

**Selection** The selection process employed uses linear based fitness ranking to assign scores to each individual in the population and roulette wheel selection to generate the intermediate population.

**Crossover** As a result of the chosen encoding scheme, crossover may not operate at the bit level as this could result in the generation of invalid gene codes. Therefore, the crossover points are restricted to specific intervals — only whole node or link values may be crossed over.

Two–point crossover is employed in this implementation. Once crossover points are selected, the gene portions are swapped. The connections within each portion remain intact, but it is necessary to adjust the connections on either side of the portion to successfully integrate it into the existing gene code. This is achieved by using node labels for each node in the network. These labels are used to identify individual nodes and to indicate the location of interconnections. Once the portion is inserted, all interconnecting links within the whole gene code are examined. If any links are now pointing to non–existing nodes, the link is changed to point to the nearest labelled node.

**Mutation** The mutation operator introduces additional noise into the genetic algorithm process thereby allowing potentially useful and unexplored regions of problem space to be probed. The mutation operator usually functions by making alterations on the gene code itself, most typically by altering specific values randomly selected from the entire gene code. In this implementation, weight mutation is employed. The operator takes a weight value and modifies it according to a random percentage in the range -200% – +200%.

## 3.5 Encoding and Decoding Schemes

Before the encoding and decoding layers can perform their respective tasks, it is necessary to arrive at a suitable encoding scheme. Many schemes were considered in preparation of these experiments, prioritising flexibility, scalability, difficulty and efficiency. These included Connectionist Encoding[6], Node Based Encoding[9], Graph Based Encoding[10], Layer Based Encoding[11], Marker Based Encoding[8], Matrix Re–writing[12, 13], Cellular Encoding[14], Weight–based encoding[3, 4] and Architecture encoding[15].

The scheme chosen is based on Marker Based Encoding which allows any number of nodes and interconnecting links for each network giving a large number of possible neural network permutations.

Marker based encoding represents neural network elements (nodes and links) in a binary string. Each element is separated by a marker to allow the decoding mechanism to distinguish between the different types of element and therefore deduce interconnections[12, 13].

In this implementation, a marker is given for every node in a network. Following the node marker, the node's details are stored in sequential order on the bit string. This includes the node's label and its threshold value. Immediately following the node's details, is another marker which indicates the start of one or more node–weight pairs. Each of these pairs indicates a back connection from the node to other nodes in the network along with the connection's weight value. Once the last connection has been encoded, the scheme places an end marker to indicate the end of the node's encoding.

The scheme has several advantages over others:

- Nodes can be encoded in any particular order, as their role within the network is determined by their interconnecting links.
- The network structures may grow without restriction—any number of nodes can be encoded along with their interconnections.
- Links between nodes can cross layer boundaries. For instance, a node in the input layer may link directly to a node in the output layer, even if there are many layers between the two.
- The system encodes individual weighting values as real numbers, which eliminates the 'flattening' of the learned weighting values which can occur when real number values are forced into fixed bit-size number values.

The decoding mechanism must take the gene codes and generate neural network data structures ready to be trained or tested. Any decoding mechanism employed must be robust and tolerate imperfect gene codes. A number of anomalies may occur after networks have been crossed over:

- Data may occasionally appears between an end marker or start marker. In such a circumstance the decoder ignores the data as it is no longer retrievable.
- It is possible that extra start or end markers be present within a node definition. In such a case, two choices are possible: either the new marker and its contents is ignored, or the previous section is ignored and the new one is taken as valid. The current implementation follows the latter approach.
- A start marker may have no corresponding end marker or vice–versa. In such a situation, the decoder ignores the entire section of the gene code.

## 4 Experiments

The problem set employed for these experiments was 5–bit parity. Each network was exposed to 5–bit patterns and trained to determine the parity of each pattern. The number of training iterations was also varied from 0, 10 and 100 iterations. The crossover rate was set to 0.75 and the mutation rate to 2%. Three experiments were carried out in total with 500 networks in each generation for 600 generations. The general aim of these experiments was twofold: to demonstrate the validity of the simulator and to ascertain how much the training or learning process affects each population's fitness. The experimental results are shown in figure 2.

**No training** When the genetic algorithm is used in isolation without the help of the learning process, the population's fitness shows very little improvement in the first 200 generations. There is then a slight increase in fitness leading to further stagnation at around the 0.3 fitness level. The genetic algorithm alone is unable to generate a successful population for this problem set.

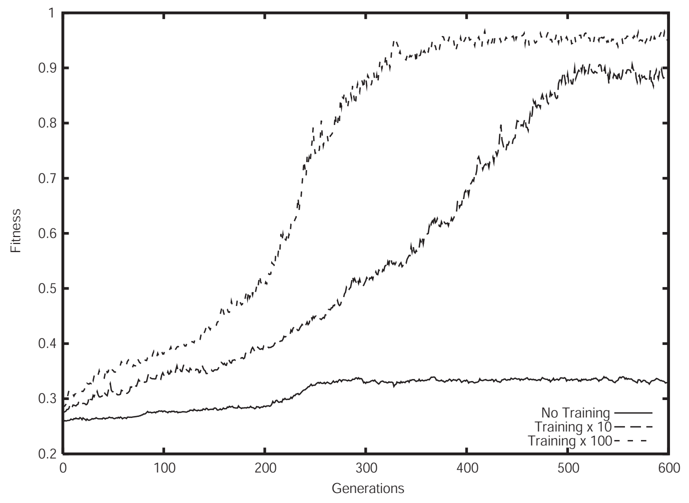

**Fig. 2.** *Population Fitness For 0-100 Training cycles*

**10 Training Iterations** The addition of training shows that even a modest increase in the population's individual learning capability, enables the simulator to achieve very high levels of fitness. The fitness level ascends steadily to 0.85 before leveling out at nearly 0.9. At this level of fitness most individuals in the population are capable of solving all 32 solutions in the 5–bit parity problem.

**100 Training iterations** Once the networks receive more training, the advantages of the training process become obvious. The population's fitness increases along a steep curve before jumping 0.3 points in 100 generations. The curve then levels out at around 0.95 - the highest level of fitness attained in this experiment set.

## 5 Conclusion

The results achieved with the simulator seem to indicate that population learning alone is not capable of solving problems of a certain difficulty. Once lifetime learning is introduced, the training process guides the population towards very high levels of fitness. The fact that the population is capable of acheiving such high levels from little training (in the case of the 10 Training iterations experiment) shows that this approach should be capable of solving more complex problems.

## 6 Acknowledgements

This research is funded by the Irish Research Council for Science, Engineering and Technology.

## References

1. J. H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor MI: The University of Michigan Press, 1975.
2. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, MA, Addison-Wesley, 1989.
3. R. S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proc. of 8th Annual Conf. of the Cognitive Science Society*, pages 823–831, 1986.
4. John F. Kolen and Jordan B. Pollack. Back propagation is sensitive to initial conditions. In Richard P. Lippmann, John E. Moody, and David S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 860–867. Morgan Kaufmann Publishers, Inc., 1991.
5. Peter J. Angeline, Gregory M. Saunders, and Jordan P. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, January 1994.
6. Richard K. Belew, John McInerney, and Nicol N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 511–547. Addison-Wesley, Redwood City, CA, 1992.
7. B. MacLennan. Synthetic ethology: An approach to the study of communication. In *Artificial Life II: The Second Workshop on the Synthesis and Simulation of Living Systems, Santa Fe Institute Studies in the Sciences of Complexity*, pages 631–635, 1992.
8. David Moriarty and Risto Miikkulainen. Discovering complex othello strategies through evolutionary neural networks. *Connection Science*, 7(3–4):195–209, 1995.
9. David W. White. *GANNet: A genetic algorithm for searching topology and weight spaces in neural network design.* PhD thesis, University of Maryland College Park, 1994.
10. J. C. F. Pujol and R. Poli. Efficient evolution of asymmetric recurrent neural networks using a two-dimensional representation. In *Proceedings of the First European Workshop on Genetic Programming (EUROGP),*, pages 130–141, 1998.
11. M. Mandischer. Representation and evolution of neural networks. In R. F. Albrecht, C. R. Reeves, and N. C. Steele, editors, *Artificial Neural Nets and Genetic Algorithms Proceedings of the International Conference at Innsbruck, Austria*, pages 643–649. Springer, Wien and New York, 1993.
12. H. Kitano. Designing neural networks using genetic algorithm with graph generation system. In *Complex Systems, 4, 461-476*, 1990.
13. P. M. Todd G. F. Miller and S. U. Hedge. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference onGenetic Algorithms and Their Applications*, pages 379–384, 1989.
14. F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Centre d'etude nucleaire de Grenoble, Ecole Normale Superieure de Lyon, France, 1994.
15. John R. Koza and James P. Rice. Genetic generation of both the weights and architecture for a neural network. In *International Joint Conference on Neural Networks, IJCNN-91*, volume II, pages 397–404, Washington State Convention and Trade Center, Seattle, WA, USA, 8-12 1991. IEEE Computer Society Press.