Forcing neurocontrollers to exploit sensory symmetry through hard-wired modularity in the game of Cellz

Julian Togelius
Department of Computer Science
University of Essex
Colchester, Essex, CO4 3SQ
julian@togelius.com

Simon M. Lucas
Department of Computer Science
University of Essex
Colchester, Essex, CO4 3SQ
sml@essex.ac.uk

Abstract- Several attempts have been made in the past to construct encoding schemes that allow modularity to emerge in evolving systems, but success is limited. We believe that in order to create successful and scalable encodings for emerging modularity, we first need to explore the benefits of different types of modularity by hard-wiring these into evolvable systems. In this paper we explore different ways of exploiting sensory symmetry inherent in the agent in the simple game Cellz by evolving symmetrically identical modules. It is concluded that significant increases in both speed of evolution and final fitness can be achieved relative to monolithic controllers. Furthermore, we show that simple function approximation task that exhibits sensory symmetry can be used as a quick approximate measure of the utility of an encoding scheme for the more complex game-playing task.

1 Background

The current interest in exploring and exploiting modularity in evolutionary robotics can be understood in several ways: as a way of studying modularity in biological evolved systems, as a way of making evolution produce systems which are easier (or at least possible) for humans to understand and thus to incorporate into other human-made systems, and as a means of scaling up evolutionary robotics beyond the simple behaviours which have been evolved until now. In our opinion, these perspectives are complementary rather than exclusive.

For those interested in evolving autonomous agents for computer games, certainly the two latter perspectives are the most important. Agents will need to be able to perform complex tasks, such as serving as opponents to human players, in environments constructed for human players, and their internal structure should ideally be amenable to changes or enhancements from game constructors.

The ways in which modularity can help scaling up and make evolved solutions comprehensible is by improving network updating speed, reducing search dimensionality, allowing for reusability, and diminishing neural interference.

When a neural network is divided up into modules, the number of connections for the same number of neurons can be significantly reduced compared to a non-modular, i.e. a fully connected network. As propagating an activation value along a connection is the most frequent operation performed when updating a neural network, the time needed for updating the network can be likewise significantly reduced. This not only allows the controller to be used in time-critical operations, like real-time games, but it also speeds up evolution.

However, even if a modular network has the same number of connections as its modular counterpart, as is the case with the architectures presented in this paper, evolution can be sped up by modularity. In most encodings of neural networks, the length of the genome is directly proportional to the number of connections, but when several modules share the same specifications, the genome for a modular network might be significantly smaller than for a non-modular network with the same number of connections. This reduces the dimensionality of the space the evolutionary algorithm searches for the solution in, which can improve the speed of evolution immensely.

Neural interference (Calabretta et al. 2003) refers to the phenomenon that the interconnection of unrelated parts of a neural network in itself can hamper evolution, because any mutation is likely to set that interconnection to a non-zero value, which means that activity in these non-related parts of the network interfere with each other. A good modularisation alleviates this problem.

Finally, many problems arising in computer games and elsewhere have the property that parts of their solutions can be reused in other problems arising in similar context. An evolutionary algorithm that could reuse neural modules in the development of new solutions could cut evolution time in such circumstances.

The flipside to all this is that not every architecture is a modular architecture, and constraining your network to modular topologies means running the risk of ruling out the best architectural solutions; constraining your network to weight-sharing (reusable) modules means even more constraints, as this is optimal only when there is indeed some repeating problem structure to exploit.

Many attempts have in the past been made to achieve the benefits outlined above while minimizing the negative effects of topological constraints. Several of these attempts try to allow for the division of the neurocontroller into modules to emerge during evolution, instead of explicitly specifying the modules. For example, Cellular Encoding (Gruau 1994), inspired by L-systems (Lindenmayer 1968) grows neural networks according to information specified in graphs, allowing segments of the network to be repeated several times. Gruau's architecture

has been put to use and expanded by other researchers, such as Hornby et al. (2001) and Kodjabachian and Meyer (1995). An alternative modular encoding, called Automatically Defined Functions (Koza 1994) is used in Genetic Programming. Bongard (2003) has recently devised an encoding based on gene regulation, which is capable of producing modular designs; a good overview of approaches such as those mentioned above is given in (Stanley & Miikkulainen 2003).

However, even in these encoding schemes, human design choices arguably influence the course of evolution; some forms of modularity are more likely to evolve than others. For example, a given encoding scheme might be better suited for evolving modules that connect to each other in a parallel fashion than for evolving modules that connect together in a hierarchic fashion. At the same time, we usually don't have a theory of what sort of modularity would best benefit a particular combination of task, environment and agent. This could be why these encodings, though mathematically elegant, have failed to scale up beyond very simple tasks, at least in neural network-based approaches. Furthermore, these encodings seem very poor at expressing re-usable modules compared to languages used for expressing hardware or software designs, such as VHDL or Java respectively. To properly express modular designs, it is necessary to allow specification not only of the details of a module, but also how modules may be sensibly interconnected together, and how new module designs may be constructed from existing ones via delegation and inheritance. The concepts of evolving objects (Keijzer et al, 2001), or object-oriented genetic programming (Lucas, 2004) suggest some promising directions, but more work is needed in these areas.

We believe that the complementary approach of explicitly defining and hard-wiring modules and their interrelations could be useful in investigating what sorts of modularity are best suited to any particular problem, or problem class; knowledge which would be useful when developing new encoding schemes allowing for emergent modularity. We also believe that explicit modular definition will scale up better than any other method in use today.

Raffaele Calabretta and his colleagues have reported increased evolvability from hard-coded modularity (with non-identical modules) in different contexts, such as robotic can-collecting (Calabretta et al. 2000) and a model of the *what and where* pathways of the primate visual system (Calabretta et al. 2003), but note the conflicting findings of Bullinaria (2002).

Of special interest in our approach are cases where aspects of the problem or agent show some form of symmetry, so that identical modules can be evolved and replicated in several positions in the controller, using different inputs. Little work seems to have been done on this, but note Vaughan (2003) who evolves a segmented robot arm with identical modules, and Schraudolph et al. (1993) use tiled neural networks that take advantage of the symmetry inherent in the game Go. However the problem of playing Go is very different than playing most computer games. They also use temporal difference learning rather than evolution.

In this paper, we are comparing the results and dynamics of evolving monolithic networks (standard multi-layer perceptrons) of different sizes with those of evolving modular architectures with identical modules that exploit sensory symmetry. The evolved neural networks are compared both in terms of maximum fitness, fitness growth, and behaviour of the resulting controllers.

As a test bed, we have used the game Cellz, which has the benefit that the agent has 8-way radial symmetry. While Cellz was developed especially for testing evolutionary algorithms, the computational expense can still be prohibitive for exploring large parameter spaces. Therefore, we have constructed a simple function approximation task to have similar difficulty and demands on network architecture as the Cellz control task, but which evaluates much faster. Experiments with different network architectures were carried out first using the function approximation task, and then using Cellz, and the qualitative similarity of results using these two tasks were investigated.

2 Methods

2.1 Cellz

The game of Cellz (Lucas 2004) has been designed as a test bed for evolutionary algorithms. The game was run as a competition for the GECCO 2004 conference, and the source code is available on the web. The elements of the game are a number of cellz and a number of food particles, and the objective of the game is for the cellz to eat as many food particles as possible. A cell eats a food particle by moving over it, which increases its mass; when its mass increases over a threshold it splits into two. The food particle, upon being eaten, vanishes and reappears somewhere else on the game area. A cell moves by applying a force vector to itself, which trades some of its mass for changing its speed – the problem of movement is not trivial, having to take momentum and friction into account. Neither is the problem of deciding which food particles to go for, which in the case of only one cell is an instance of the travelling salesman problem, but quickly becomes more complex as other cells are added. A major problem is not to go for a food particle that another cell will get to first. Furthermore, each game starts with the cells and the food in random locations, and each new piece of food is added in a random location, which means that evolution should aim to acquire general good behaviours rather than those that just happen to work well for a particular game configuration. Figure 1 shows the trace of a part of a game run using an evolved perceptron controller (from Lucas 2004), and illustrates how the cells (thick lines) move in chaotic patterns in their attempts to eat food (dots) and divide.

Each cell is equipped with eight cell sensors and eight food sensors spread evenly around its body; (Figure 2) each sensor measures the distance to and concentration of other cellz or food in its 45 degree angle. The sensor

arrays are used as inputs to the controllers, and their outputs are used to generate the force vectors.

The total mass of all cells was used as fitness value for each game, which was run for 1000 time steps, and the fitness value for each individual in each generation was computed as the mean of ten such games in order to reduce noise.

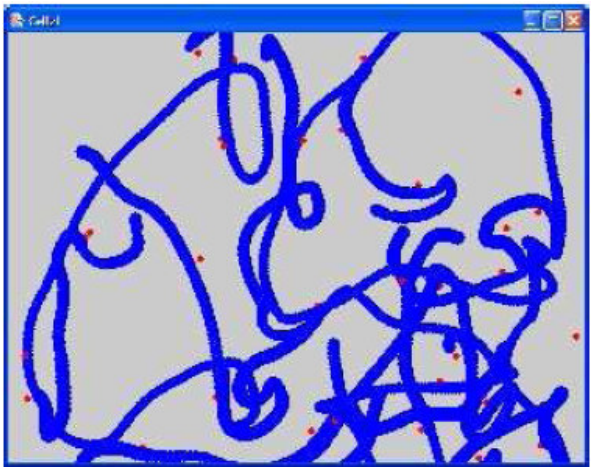

Figure 1: A sample run of Cellz with an evolved perceptron controller (from Lucas 2004).

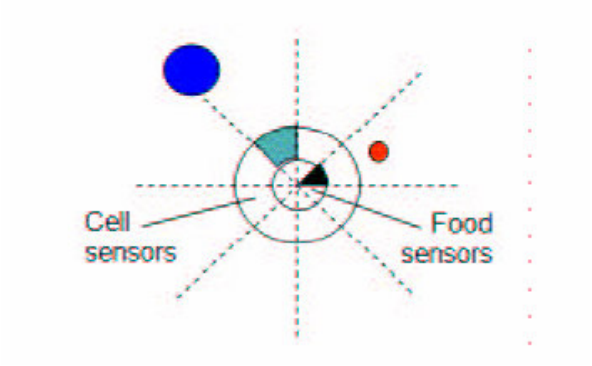

Figure 2: The wrap around input sensors. From Lucas (2004).

2.2 Neural networks

Four different neural architectures were tested and compared. In all of them, each neuron implemented a *tanh* activation function, and the synapse weights were constrained to be in the range [-1..1], as were inputs and outputs.

The first two architectures were standard multi-layer perceptrons (MLPs). The first MLP consisted of an input layer of 16 neurons, an 8 neuron hidden layer and an output layer of two neurons. The second MLP had two hidden neuron layers of 16 neurons each. In both architectures, positions 0-7 received inputs from the ”food” input vector of the Cellz agent, positions 8-15 received inputs from the ”cells” input vector, and the two outputs from the network were used to create the force vector of the cell.

The other two “convoluted” architectures consist of eight separate but identical neural network modules - they share the same genome. Each module can be thought of as assigned to its own pair of sensors, and thus being at the same angle r relative to the x axis as those sensors. The outputs from the each module’s two output units is rotated –r degrees, and then added to the summed force vector output of the controller.

In the convoluted architectures, each module gets the full range of sixteen inputs, but they are displaced according to the position of the module (e.g. module number 3 gets food inputs 3, 4, 5, 6, 7, 0, 1, 2, in that order, while the input array to module 7 starts with sensor 7; Figure 3). In the first convoluted architecture the modules lack hidden layer, but in the second convoluted architecture, each has a hidden layer of two neurons.

It is interesting to compare the number of synapses used in these architectures, as that number determines the network updating speed and the dimensionality of the search space. The MLP with 8 hidden neurons has 144 synapses, while the MLP with two hidden layers totals 544 synapses. The perceptron-style convoluted controller has 32 synapses per module, which sums to 256 synapses, and the hidden-layer convoluted controller has 36 synapses per module, which sums to 288 synapses. It should be noted that while the convoluted controllers have little or no advantage over the MLPs when it comes to updating speed, they present the evolutionary algorithm with a much smaller search space, as only 32 or 36 synapses are specified in the genome.

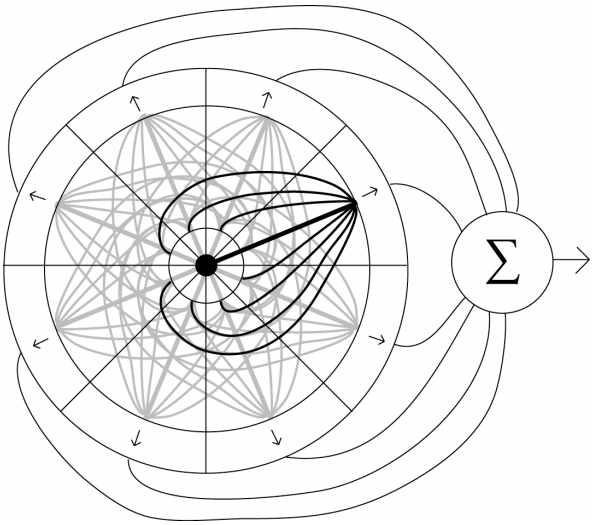

Figure 3: Simplified illustration of the convoluted architectures, taking only one type of sensor into account. The connections in black are the connections from all sensors to one module; this structure is repeated (grey lines) for each module.

2.3 Function approximation task

Like the Cellz task, the function approximation task requires the network to have 16 inputs and 2 outputs. The input array is divided into two consecutive arrays of 8 positions; each position has an associated angle in the same manner as the Cellz controller. Each time a network is evaluated, a random position on an imaginary circle, i.e. a random number in the range [0..2π], is produced. The network inputs receive activations corresponding in a nonlinear fashion to their associated angles' distance to the target position. The function to be approximated by the outputs of the network is the sine and cosine of the target position, and the fitness function is the mean absolute summed difference between these values and the actual network outputs. The time it takes to evaluate a neural network on the function approximation task is on the order of a thousand times less than the time taken to evaluate the same network as a neurocontroller for Cellz.

2.4 Evolutionary algorithm

Controllers for the agents were evolved using an evolutionary algorithm with a population size of 30. Truncation selection was used, and elitism of 5; at each generation, the population was sorted on fitness, the worse half was replaced with clones of the better half, and all controllers except the top 5 were mutated. Mutation consisted of perturbing all synaptic weights by a random value, obeying a Gaussian distribution with mean 0 and standard deviation 0.1.

3 Results

In all the graphs presented in this section, the dark line tracks the fitness of the best controller in each generation, while the other line represents the mean population fitness.

3.1 Evolving function approximators

For the function approximation problem, we define fitness to be the negative of the mean error – which gives a best possible fitness of zero. Each figure in this sub-section depicts the mean of ten evolutionary runs. Both monolithic (MLP) architectures were evolved for 100 generations. (Figures 4 and 5) They eventually arrived at solutions of similar quality, though the dual-layer MLP took longer time to get there.

The perceptron-style convoluted network reached fitness similar to that of the monolithic networks, but somewhat faster (Figure 6). The real difference, though, was with the convoluted network with one hidden layer; it achieved much higher fitness than any of the three other architectures, and did so with fewer fitness evaluations (Figure 7). In this case, we are observing a very clear benefit of the enforced modular structure. Next we investigated how well this function approximation task serves as a test-bed for the real game, which has a significant degree of noise, together with complex dynamics. While the network weights to solve each task are likely to be very different, the overall connection topologies, and the constraints on those topologies are likely to be rather similar, based on our construction of the function approximation task. Hence, while the evolved weights cannot be transferred from the function approximation task to the game, it is still possible that the both the task and the game measure similar qualities of the encoding schemes.

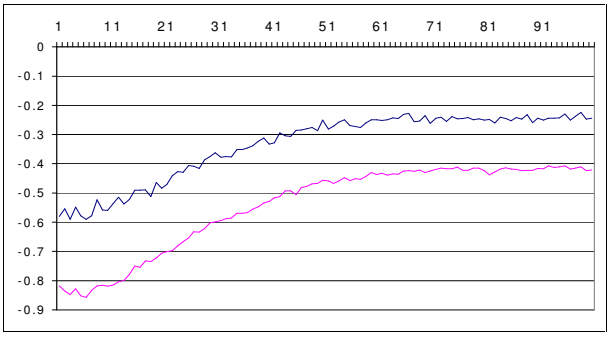

Figure 4: MLP with one hidden layer on the function approximation task.

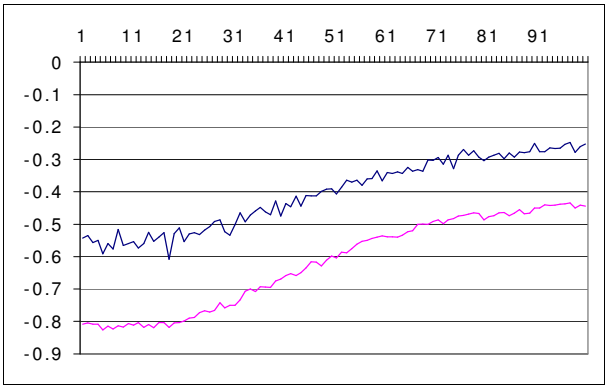

Figure 5: MLP with two hidden layers on the function approximation task.

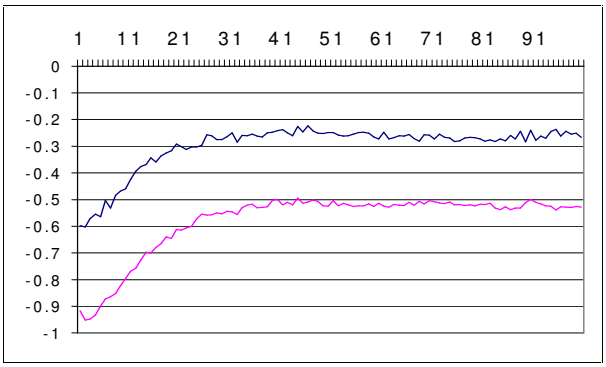

Figure 6: Perceptron-style convoluted network on the function approximation task.

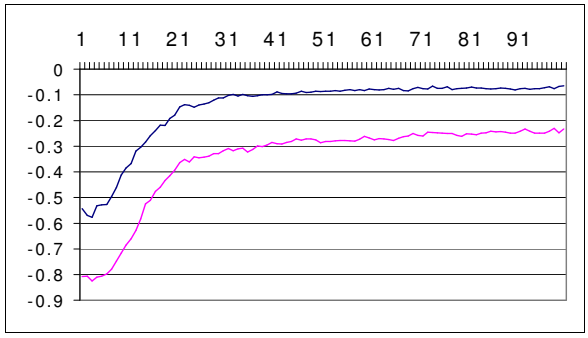

Figure 7: Convoluted network with one hidden layer on the function approximation task.

3.2 Evolving Cellz controllers

The two MLP architectures were evolved for 200 generations. Each figure now shows a single run, but each experiment was repeated several times, and the graphs shown are representative. The one-layer MLP evolved somewhat faster and reached a higher final fitness. Both evolutionary runs produced good controllers, whose agents generally head straight for the food, even though they fairly often fail to take their own momentum into consideration when approaching the food, overshoot the food particle and have to turn back. (Figures 8 and 9).

Finally, the two convoluted controllers were evolved for 100 generations, and quickly generated very good solutions. The convoluted controller with a hidden layer narrowly outperformed the one without. Not only did good solutions evolve considerably faster than in the cases of the MLPs, but the best evolved convoluted controllers actually outperform the best evolved MLP controllers with a significant margin. As the computational capabilities of any of the convoluted controllers is a strict subset of the capabilities of the MLP with two hidden layers, this is slightly surprising, but can be explained with the extravagantly multidimensional search problems the MLPs present evolution with – even if a better solution exists it is improbable that it would be found in reasonable time.

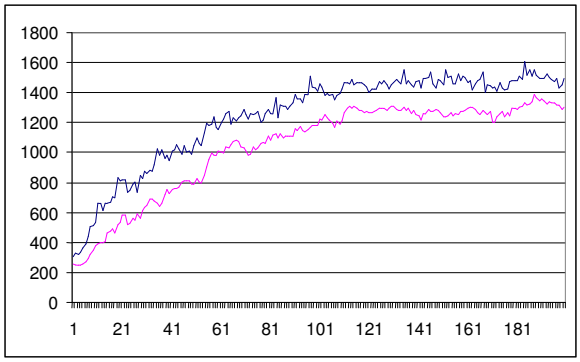

Figure 8: Evolving an MLP with one hidden layer for the Cellz game.

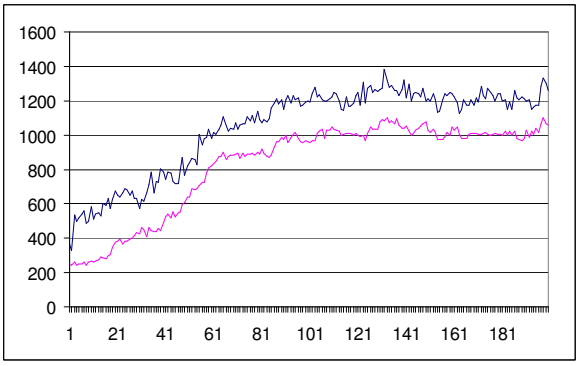

Figure 9: Evolving an MLP with two hidden layers for the Cellz game.

It is also interesting to note that the length of the neural path from sensor to actuator in the robots (that is, the number of hidden layers) seems to be of relatively small importance. (Figures 10 and 11) A comparison between controllers evolved in this paper, the winner of the GECCO 2004 Cellz contest, and the hand designed controllers mentioned in the original Cellz paper (Lucas 2004) is presented in Table 1. Note that the differences between the best three controllers are not statistically significant. The convoluted controller with one hidden layer is one of the best controllers found so far (though the convolutional aspect of the controller was hand-designed). Note that the winner of the GECCO 2004 contest was also partially hand-designed, as a neural implementation of the hand-coded sensor controller[1]. So far the purely evolved neural networks have been unable to compete with the networks that have been partially hand-crafted.

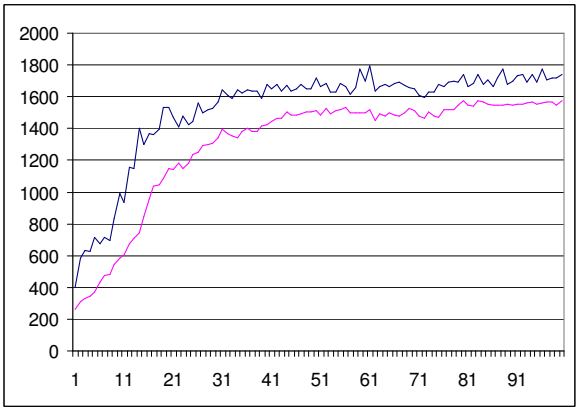

Figure 10: Evolving a perceptron-style convoluted controller for the Cellz game.

[1] See: http://cswww.essex.ac.uk/staff/sml/gecco/results/cellz/CellzResults.html

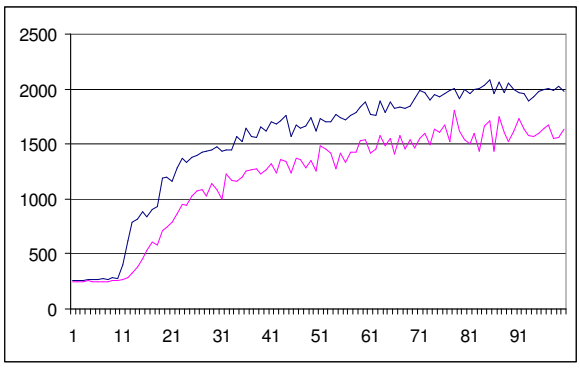

Figure 11: Evolving a convoluted MLP controller with one hidden layer for the Cellz game.

Controller	Fitness	S. E.
JB_Smart_Function_v1.1 (Winner of GECCO 2004 Cellz contest)	1966	20
Convoluted controller with one hidden layer	1934	18
Hand-coded sensor controller	1920	26
MLP with one hidden layer	1460	13
Hand-coded greedy controller	1327	24
MLP with two hidden layers	1225	14

Table 1: Mean fitness and standard errors over 100 game runs for controllers mentioned in this paper in comparison to other noteworthy Cellz controllers.

4 Conclusions

Our results clearly show that hard-coding modularity into neurocontrollers can increase evolvability significantly, at least when agents show symmetry, as they do in many computer games and robotics applications. They also show that certain types of modularity perform better than others, depending on the task at hand. Adding hidden neural layers might either increase or decrease evolvability, depending on the task. As neural encodings that intend to let modularity emerge always have certain biases, these results need to be taken into account when designing such an encoding.

We have also seen that the performance of the different architectures on the fitness approximation task are qualitatively comparable to the results of the same networks on the full Cellz task, e.g. the MLP with two hidden layers evolves more slowly than the MLP with only one, and the convoluted networks outperform monolithic networks. This suggests that this much simpler task can be used to further investigate the merits of different network architectures for Cellz; in particular, it might be possible to evolve network architectures for this simpler task, and later re-evolve the connection strengths of the evolved architectures for the Cellz task. The advantage of using the simpler task as a test-bed is that it is around 1,000 times faster to compute. This method can probably be used for other games as well.

The research described here is part of the first author's doctoral project investigating the role of modularity in artificial evolution; previous work on evolving layered structures was reported in Togelius (2004). In the future, we plan to extend this approach to more complicated tasks and input representations, such as first-person games with visual input. Eventually, we aim towards using the results of those studies as a requirements specification in the creation of new representations with which to evolve modular systems.

Bibliography

Bongard, J. C. (2003): *Incremental Approaches to the Combined Evolution of a Robot's Body and Brain*. PhD dissertation, University of Zurich.

Bullinaria, J. A. (2002): *To Modularize or Not To Modularize?* Proceedings of the 2002 U.K Workshop on Computational Intelligence: UKCI-02.

Calabretta, R., Nolfi, S., Parisi, D., Wagner, G. P. (2000): *Duplication of modules facilitates functional specialization.* Artificial Life, 6(1), 69-84.

Calabretta, R, Di Ferdinando, A. D., Wagner, G. P., Parisi, D. (2003): *What does it to take to evolve behaviorally complex organisms?* Biosystems 69(2-3): 245-62.

Gruau, F. (1993): *Genetic Synthesis of Modular Neural Networks*. In Forrest, S. (Ed.): Proceedings of Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, 318-325.

Hornby, G. S., Lipson, H., Pollack, J. B. (2001): *Evolution of Generative Design Systems for Modular Physical Robots.* IEEE International Conference of Robotics and Automation.

M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer (2001) *Evolving Objects: a general purpose evolutionary computation library*, Proceedings of Evolution Artificielle. Lecture Notes in Computer Science 2310, 231-244.

Kodjabachian, J. and Meyer, J. A. (1995): *Evolution and development of control architectures in animats.* Robotics and Autonomous Systems, 16, 161-182.

Koza, J. R. (1994): *Genetic Programming II: Automatic Discovery of Reusable Programs.* Bradford Books.

Lindenmayer, A. (1968): *Mathematical models for cellular interaction in development: Parts I and II.* Journal of Theoretical Biology, 18, 280-299, 300-315.

Lucas, S. M. (2004): *Cellz: A Simple Dynamic Game for Testing Evolutionary Algorithms.* Proceedings of the Congress on Evolutionary Computation, 1007-1014.

Lucas, S.M. (2004), *Exploiting Reflection in Object Oriented Genetic Programming*, Proceedings of the European Conference on Genetic Programming, 369-378.

Schraudolph, N. N., Dayan, P., Sejnowski, T. J. (1993): *Temporal Difference Learning of Position Evaluation in the Game of Go.* In Cowan et al. (eds): Advances in Neural Information Processing 6. San Mateo, CA: Morgan Kaufmann, 817 – 824.

Stanley, K. O. and Miikkulainen, R. (2003): *A Taxonomy for Artificial Embryogeny*. Artificial Life, 9(2), 93-138.

Togelius, J. (2004): *Evolution of a subsumption architecture neurocontroller.* Journal of Intelligent and Fuzzy Systems 15(1), 15-20.

Vaughan, E. (2003): *Bilaterally symmetric segmented neural networks for multi-jointed arm articulation.* Unpublished paper, University of Sussex. Available at: http://www.droidlogic.com/sussex/adaptivesystems/Arm.pdf