Theory Biosci. (2008) 127:187–194
DOI 10.1007/s12064-008-0029-9

ORIGINAL PAPER

Neuroevolution and complexifying genetic architectures for memory and control tasks

Benjamin Inden

Received: 31 August 2007 / Accepted: 18 November 2007 / Published online: 16 April 2008
© Springer-Verlag 2008

Abstract The way genes are interpreted biases an artificial evolutionary system towards some phenotypes. When evolving artificial neural networks, methods using direct encoding have genes representing neurons and synapses, while methods employing artificial ontogeny interpret genomes as recipes for the construction of phenotypes. Here, a neuroevolution system (neuroevolution with ontogeny or NEON) is presented that can emulate a well-known neuroevolution method using direct encoding (neuroevolution of augmenting topologies or NEAT), and therefore, can solve the same kinds of tasks. Performance on challenging control and memory benchmark tasks is reported. However, the encoding used by NEON is indirect, and it is shown how characteristics of artificial ontogeny can be introduced incrementally in different phases of evolutionary search.

Introduction

The work described here is a new approach in the area of neuroevolution, which is based on the following approach: Artificial neural networks, as a very simple abstraction of animal nervous systems, are used to control robots or other agents that have to solve some given task. As it is difficult to design neural networks with a suitable topology and connection weights by hand, evolutionary algorithms are used to find networks that show the same type of robust and flexible behavior that can also be observed in animals.

B. Inden (✉)
Max Planck Institute for Mathematics in the Sciences,
Inselstr. 22, 04103 Leipzig, Germany
e-mail: inden@mis.mpg.de

Many tasks require that—or at least are much easier if—the agent has internal memory; if there is no memory, objects out of sensor range cannot influence behavior. Neural networks can implement memory through recurrent connections. In existing work on evolutionary robotics, often fixed neural network topologies are used (Nolfi and Floreano 2000); choosing the topology in advance, however, is difficult: if the chosen topology is too small, there may not be enough memory and processing elements to solve the task; if the chosen topology is too large, learning correct connection weights may take too long. Fixed architectures are also a problem for incremental evolution, where newly evolving features may interfere with features already evolved. Therefore, evolutionary algorithms that can also evolve the topology of neural networks are desirable. The well-known neuroevolution of augmenting topologies (NEAT) (Stanley and Miikkulainen 2002, 2003) is such a method. In the section "The NEON method", a new method based on NEAT will be introduced. This method also employs the principles of artificial ontogeny, which we discuss first.

Artificial ontogeny

The imitation of the natural process of development for artificial life is called artificial ontogeny (Bongart 2003), or artificial embryogeny (Stanley and Miikkulainen 2003). This method is typically used together with an evolutionary algorithm, and entails a growth process, where a mature phenotype is constructed from a simple initial state using information from the genotype. Several researchers have already used artificial ontogeny to construct neural networks for robot control tasks (Eggenberger 1996).

Using artificial ontogeny instead of a direct encoding in connection with neuroevolution may have several advantages: First, compressible phenotypes can be encoded in a more compact genotype through gene reuse. This enhances scalability of neuroevolution (Roggen and Federici 2004). Second, the growth process can exploit constraints from the environment. It can evolve to be adaptive: produce different phenotypes in different environments such that each is adapted. Third, restructuring the developmental process by evolution makes linkage learning and coordinated variability for phenotypic variables possible (Toussaint 2003).

On the other hand, existing artificial ontogeny systems seem to be biased towards phenotypes of low complexity (high compressibility), and have considerable difficulties evolving high complexity phenotypes (Harding 2006). Simulating ontogeny is also very time consuming.

Artificial ontogeny may be ultimately advantageous in incremental evolution scenarios, where the additional cost of learning a good representation in the beginning pays later through coordinated variability and gene reuse.

The aim of the work reported here is to use neuroevolution to solve challenging benchmark tasks *and* make use of complex genetic architectures (that include features of developmental processes) that can be used in incremental evolution scenarios to solve increasingly complex tasks with many inputs and outputs. In order to achieve this goal, features of artificial ontogeny are introduced incrementally, starting from a mode of evolutionary search that is equivalent to using a direct encoding. Also, while artificial ontogeny methods are typically immediately evaluated using tasks with large numbers of inputs and outputs (but not too difficult in terms of fine control or memory requirements), here we start evaluating with benchmark tasks typically used for direct encodings, and then show *in principle* how large neural networks can be encoded by rather small genomes. Ultimately, of course, direct demonstration that the method *also* works for tasks with large input and output spaces is desired, but solutions to these tasks are meant to arise by incremental evolution from small networks. As a side effect of this approach, we gain some more insight about how NEAT-like methods find solutions, and study a new benchmark task on memory evolution.

The following practical considerations have been made in devising the proposed encoding: First, a good level of abstraction has to be used. In many cases, the results of processes [e.g., gradients or cells (Stanley and Miikkulainen 2003)] can be created directly instead of simulating the processes. This may take away some possibilities which evolution could exploit, but the speedup is essential.

Second, stochasticity has to be limited, otherwise large population sizes or multiple evaluations of a single genotype are necessary. Instead, one could provide access to large amounts of *unchanging* random data, which can be exploited to construct the phenotype.

Third, representations and operators have to be designed such that heritability is high enough; while some mutations might cause large changes on the phenotype layer, there must still be enough mutations that cause slight changes only.

The NEON method

Evolving neural network topology poses some technical problems. If nodes are just added without connections to existing nodes, the problem of bloat arises; but if they are attached to existing network structure, they may easily disrupt function. NEAT (Stanley and Miikkulainen 2002, 2003) is a neuroevolution system which deals with these problems quite successfully. On the one hand, operators are designed such that disruption of existing function is less likely; on the other hand, networks with topological innovation are protected against extinction by a speciation mechanism based on neural network similarity, which is easily calculated by assigning historical markings called "global innovation numbers" to new connections or nodes. The "neuroevolution with ontogeny" (NEON) system used for the simulations described here can emulate NEAT if parameters are set correspondingly, and was designed to enable incremental introduction of complex genetic architecture using developmental processes (Inden 2007). The mutation operators used by NEAT are available in NEON as operators of developmental change. Similarly, the NEAT speciation mechanism can be applied on the final phenotype in NEON. When emulating direct encoding, mutations always insert new genes (encoding one developmental operation each) at the end of the developmental sequence. This is equivalent to just applying the mutation directly as in NEAT, the difference being that these operations must be "replayed" every time the phenotype is constructed. If other types of mutations are used, the evolutionary order and the developmental order of the operations may diverge, and genes may encode several operations. Each gene can access arbitrary amount of data by accessing a data stream, which is in fact just a chunk of output from a random generator function, and is referenced by the gene through specifying a seed for the random number generator. This data is therefore *unchanging* provided the same key is used every time. Because of the dependence of the phenotype on this data, NEON could be classified as using external ontogeny in the sense of Bentley and Kumar (1999), although with a practically unlimited reservoir of external patterns. In contrast, explicit embryogeny is a method that uses sequential programs composed of actions (possibly with programming

constructs like branches, loops, and subfunctions) as genotype [cellular encoding (Gruau 1994) is an example of a method which uses explicit embryogeny to construct neural networks]; implicit embryogeny uses sets of rules that are applied repeatedly if their preconditions are matched to build the phenotype. These rules are often given in the form of artificial genetic regulatory networks (Eggenberger 1996; Bongart 2003), or in the form of a neural network (Federici 2005).

After having described the basic idea of the NEON method, what follows is a more detailed description. Standard sigmoid neurons are used for the neural network; their transfer function is

$$o_i(t+1) = \frac{2}{1+\exp\left(-2\left(\sum_{j\in N} w_{ij}o_j(t)\right)\right)} - 1$$

[which is equivalent to the tanh function for the weighted input sum, and is in the range $(-1,1)$]. Connection weights are in the range $(-3,3)$. The network consists of a single output node connected to all inputs when the developmental process begins.

The developmental operations used in NEON are the following:

- A specified fraction of the connection weights are perturbed randomly, each with a value drawn from a normal distribution
- A specified fraction of the connection weights are set to random values from the range of allowed weights
- Two neurons are randomly chosen and connected if no connection exists between them
- A neuron is randomly chosen and a recurrent connection established if none exists yet
- An existing connection is chosen and a neuron inserted in between. The connection weight to the new neuron is set to 1.0, while the connection from the new neuron is assigned the weight of the old connection. The old connection is disabled
- A disabled connection is enabled
- The state of a connection is toggled (enabled/disabled).

All parameters for the developmental operations are taken from the data stream. Instead of the innovation numbers used by NEAT, NEON assigns tags to each neuron and connection. These tags are also taken from the data stream and are unique with very high probability.

All developmental operations require choosing nodes or connections. A naive implementation would be to choose according to their position in a list. But when a new connection or node is created, all operations later in developmental time would then work on changed lists, which would make this operation a macromutation. Therefore, a more sophisticated method is used which chooses nodes or connections based on matching their tags to parameters of the developmental operations. The goal is to make choosing each item roughly equally likely, and have insertion of a new tag change very few, if any, subsequent choices.

The tags for all input and output neurons are directly taken from the data pool accessed by the gene with the lowest time index. All other tags are computed from these tags, which ensures that identical topological innovations get assigned identical tags.

Each gene has an id number, a key to access the data pool, a volume field that specifies how many developmental operations to read from the pool, and a time index that specifies where, in the sequence of developmental operations, the gene applies (Fig. 1).

Insertion of a gene usually happens with its volume set to 1, while probabilities can be given for the time index of the new gene being below or above the current highest time index in the genome. Among the substitution operations that can be applied to a gene are increment and decrement of the volume, and change of key. The deletion operator removes a gene completely.

The population is partitioned into species every generation. An individual is assigned to a species if it is sufficiently similar to that species' representative from the previous generation. If it cannot be assigned to an existing species, a new species is created. Each species' offspring size is made proportional to its mean fitness; this prevents a slightly superior species from taking over the whole population. Inside the species, the worst performing individuals are deleted, after which stochastic uniform selection is used for the rest. Species with offspring size greater than five also keep their best performing individual. If a species' maximum fitness has not increased for more than 200 generations and it is not the species containing the best network, its mean fitness is multiplied by 0.01, which usually means it dies out. Like in the SharpNeat implementation (Green 2006), NEON

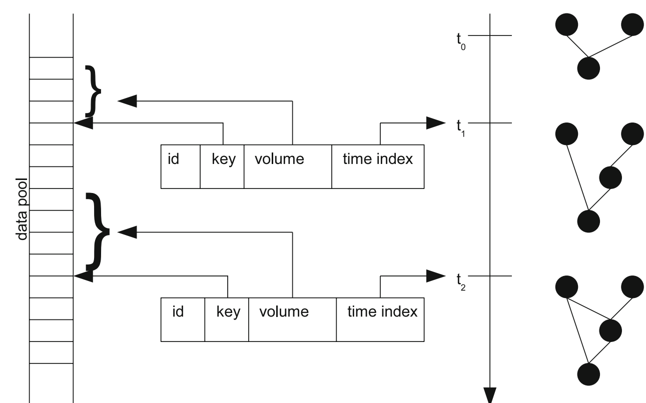

Fig. 1 Ontogeny with the neon method. In the *middle* are the genes, while on the *right*, the sequence of phenotypes is depicted

checks if the number of species is in some desirable range, typically $n_{spec} = 35...45$; if not, the similarity threshold for speciation is adjusted.

Controller evolution

As discussed above, we first benchmark NEON when emulating direct encoding (i.e., NEAT). A challenging nonlinear benchmark task is double pole balancing, where two poles of different lengths, which are mounted on a cart that can drive back and forth on a track, must be kept upright. The basic measure of performance is the number of time steps that the cart stays within certain distance from its point of origin, and both poles do not deviate from the upright position by more than some angle. In the simpler Markovian version of the task (DPV), the neural network gets the pole angles and angular velocities, as well as the cart position and speed, as input. A bias input is also provided. In the more difficult non-Markovian version (DPNV), all velocity inputs are missing. There is also a slightly more difficult version of DPNV (AWDPNV), where wiggling of the poles is punished and generalization to at least 200 out of 625 starting angles is required. All tasks have been described in more detail in Wieland (1991), Stanley and Miikkulainen (2003), and Stanley (2004).

A series of experiments with 30 runs each are reported here (comparisons were done using a Wilcoxon rank sum test on the number of evaluations; unless reported otherwise, a run lasted for 500 generations at most).

NEON can solve the DPV task using 5,628 evaluations on average (NEAT 3,600), the DPNV task using 49,918 evaluations (NEAT 20,918), and AWDPNV using 51,588 evaluations, final solutions solving 252 of the 625 tasks on average [NEAT 24,543 evaluations, 286 tasks, as reported in Stanley and Miikkulainen (2002) and Stanley (2004)]. So NEON finds solutions using the same order of magnitude of evaluations as NEAT, although somewhat slower. Upon inspection of the NEAT source code, one finds that the perturb operation is very sophisticated there, also making a distinction between connections that arose earlier and those that arose later in evolutionary history. Such a distinction was not attempted to be made in NEON, where different time axes for ontogeny and phylogeny complicate the issue. In any case, the performance is sufficient for studying how difficult tasks can be solved using indirect encodings, and could probably be increased by further tuning on the parameters.

The reported number of evaluations were reached with a standard configuration using a population size of 150, where the developmental operations were applied with the following probabilities: weight perturbation 82.4%, weight setting 10%, connect 5%, connect recurrent 0.5%, split 0.1%, toggle enable 1%, re-enable 1%. Perturbations and weight settings each affected 40% of the connections on average. Perturbations added a value drawn from a normal distribution with standard deviation of 0.24 to each affected weight. For further studies, a similar configuration, but with a population size 1,000 (called DL1), was used because it proved more robust to parameter changes. This configuration used, on average, 108,126 evaluations for DPNV, and 108,800 for AWDPNV. A more detailed description of parameters and results for these experiments can be found in Inden (2007).

For comparison, a simple tournament selection setup with population size 1,000, tournament size 2, and elite size 10 can solve DPNV using 156,826 evaluations on average; it also finds solutions in 73% of the runs for AWDPNV. Although both results are significantly worse than those for DL1, in this case the gap between speciation selection and standard selection methods is not too large once good operators are used. This may be also due to the way the split operation is designed: as can be seen in Table 1, it has positive effects in 5% and neutral effects in 18% of the cases, which means that disruptive effects can indeed be avoided relatively often. According to this data, it can also be seen that the immediate fitness effects of set weight operations are more often negative than those of the perturb operation [which is expected because it is a "macromutation"]. Nevertheless, further experiments have shown that configurations without the set weight operation perform significantly worse; this operation is necessary for creating and maintaining enough diversity in the population. The fraction of neutral connect operations in the table is large because the operation by default has no effect if the randomly chosen nodes are already connected. What is also remarkable is the rather large fractions of toggle enable/re-enable operations with positive effects. Indeed, setups without both of these operations perform significantly worse.

Evolution of complex genetic architectures

Now we show in principle how large phenotypes can be achieved with small genotypes in NEON. Table 2 lists

Table 1 Immediate fitness effects of adding developmental operations when original fitness is at least 60 time steps (DPNV, configuration DL1, 30 runs)

Operation	Negative	Neutral	Positive
Perturb	0.774	0.125	0.101
Set weight	0.926	0.061	0.013
Connect	0.290	0.686	0.024
Split	0.768	0.180	0.052
Toggle/re-enable	0.738	0.191	0.070

Table 2 Performance of NEON with complex genetic architectures

Configuration	Changes from DL1	Mean evaluations	Comparison
DL2	Key substitution (0.05) Deletion (0.45) Volume increment (0.2) Volume decrement (0.05)	294,928.1 (2.7× DL1)	Significantly worse ($p = 9.917 \times 10^{-7}$)
DL3	Key substitution (0.1) Deletion (0.15) Volume increment (0.2) Volume decrement (0.05)	137,571.6 (1.3× DL1)	Significantly worse ($p = 0.01759$)
DL4	Lower time index: $p = 0.2$	129,142.6 (1.2× DL1)	Indistinguishable ($p = 0.3738$)
DL5	Lower time index: $p = 0.4$	170,784.3 (1.6× DL1)	Significantly worse ($p = 0.004994$)

results for a number of experiments where other mutations, besides the insertion at the end of the developmental sequence, were allowed. These mutations lead to divergence of ontogenetic and phylogenetic trajectories; they can also make genes coding for arbitrary amounts of developmental operations. The search performed through phenotype space is no longer equivalent to the search performed by NEAT with its direct encoding; search also becomes less efficient. For example, by incrementing the volume of a gene (the number of developmental operations its reads from the data pool), only the particular operation that comes next in the data pool can be added, which may lead to repeated exploration of the same phenotypes. Also, the tagging system described above does not eliminate *all* side effects that inserting a developmental operation has on subsequent developmental operations.

Configurations DL2 (run for at most 1,000 generations) and DL3 use different mixtures of mutations. The advantage over the standard configuration is that they lead to smaller genomes: DL2 solutions had from 16 to 37 genes (mean 26.3), while DL3 solutions had from 31 to 108 genes (mean 64.8). For comparison, DL1 solutions had between 33 and 176 genes, the mean being 89.1.

Configurations DL4 and DL5 show that performance also degrades slightly but significantly when the probability of inserting a new operation not at the end of, but somewhere within the sequence, is increased.

Above, the idea was mentioned that developmental encodings may be most useful in an incremental evolution scenario, where structure for a newly evolving task is first stored uncompressed in the genome, and later reorganized and compressed as that feature gets conserved. To study this idea in the context of NEON, the solutions to the DPNV problem were taken from the 30 runs of standard configuration DL1.

These solutions were then evolved for 500 more generations using a different fitness function. This function, as before, counted the number of balancing time steps, but only up to a maximum of 1,000 time steps for saving run time. That number was multiplied with a function that rewards smaller genomes linearly. The mutation probabilities now were like in experiment DL2.

The compressed solutions of these runs were then re-evolved to reach 100,000 time steps, either with mutation probabilities as in DL2, or with only insertions allowed as in DL1. The length of the genomes were not evaluated in these runs.

After the compression runs, the solutions had between 10 and 30 genes, the mean being 17.1. After the first kind of re-evolution, the respective values were 11, 30, and 18.4; after the second kind of re-evolution, they were 11, 39, and 19.5. All re-evolution runs re-reached 100,000 time steps. Re-evolution took 5.2 generations and 5,181 evaluations on average with the first method, or 3.5 generations and 3,527 evaluations on average with the second method (2 of the 30 compressed solutions did not need any re-evolution to reach 100,000).

This means that the strongest compression method achieves compression to 19% of the original size on average. On examination of the re-evolved solutions, one can find that the volume of the kernel genes is 1.64 on average, that is, each gene reads on average 1.64 developmental operations from the pool (the highest volume found is 6).

Memory evolution

Now we return to the emulation of direct encoding, but study another problem. As a benchmark task for memory evolution, pole balancing is not satisfying, because on the one hand, memory requirements to solve this task are limited, and on the other hand, it is also a challenging nonlinear control problem; so, in a sense, two difficult things are benchmarked at once.

Table 3 Fraction of successful NEON runs, mean number of evaluations and generations in successful runs, and mean highest fitness in unsuccessful runs for several sequence recall tasks

Task	Success	Evaluations	Generations	Fitness	Success	Evaluations	Generations	Fitness
1 Bit strict	1.0	10,979	11.1	–	1.0	24,661	24.9	–
2 Bit strict	1.0	47,499	49.3	–	0.83	183,398	185.2	14.0
3 Bit strict	0.9	282,426	294.9	47.9	0.17	399,376	403.4	43.8
4 Bit strict	0.0	–	–	61.9	0.0	–	–	58.6
1 Bit relaxed	1.0	3,494	3.5	–	1.0	1,033	1.0	–
2 Bit relaxed	1.0	16,921	17.2	–	0.77	205,198	207.2	14.0
3 Bit relaxed	1.0	70,573	73.4	–	0.0	–	–	40.8
4 Bit relaxed	0.23	284,865	297.6	61.6	0.0	–	–	53.2

Left group of columns: speciation selection (configuration DL1); Right group: rank selection

Road sign problems (Rylatt and Czarnecki 2000) are a class of tasks that are to be solved with internal memory. Particularly well known is the T maze task, where the robot has to drive through a narrow corridor. At some point, there is a light signal coming from one of the two lateral directions. The robot has to continue to drive through the corridor to a fork, where it has to turn in the direction of the light signal, which is, however, no longer visible at that point. If it does so, it will reach a reward area. It has been reported, however, that robots can solve the task without memory by driving closer to one wall as soon as the signal appears, and later turn into the direction of the closer wall (Ziemke et al. 2004). So this task in its standard version cannot be used for benchmarking memory evolution.

The sequence recall tasks presented here are a family of tasks that can be thought of as abstractions of road sign problems to the recall of bit sequences. In these tasks, a neural network with just one input (plus bias input) and one output is needed. Parameters are a period length ω (typically 10 time steps), the number of periods n, a readout offset r, and the number of episodes; bit strings are chosen randomly for each episode, and these assignments are usually kept fixed for the whole run. In each episode, the n bits are presented to the network sequentially during the first halves of the periods. The network output is checked at the end of each period, but subject to a shift specified by the readout offset. When no signal is presented to the network, it just gets the middle value of the input range (usually 0.0) as input. The closer the readout is to the complement of the input of r steps ago, the higher the fitness contribution. The total fitness is just the sum of all fitness contributions.

A special case is to test recall only after the whole bit sequence has been presented. In this study, all possible binary sequences for the 1–3 bit tasks are presented in separate episodes. For more than 3 bits, only eight different sequences are tested in separate episodes to save run time. As a fitness contribution of 2.0 can be gained for every correctly remembered (and inversely output) bit (this is the difference between the minimum and the maximum of the output range), the following maximal fitness values can be achieved: 1 bit 4.0, 2 bit 16.0, 3 bit 48.0, and 4 bit 64.0. A task was considered solved when the fitness of the best network differed from the respective value by not more than 0.01.

The last point, however, deserves further consideration. Does the network really have to approximate the expected output that closely to demonstrate that it remembered the input sequence? It could easily happen that we again benchmark on two things simultaneously, the ability to memorize and the ability to produce correct output signals. To better understand how difficult memorizing actually is, it has also been combined with an easier output producing task: an output received the full fitness score of 2.0 if it did not deviate from the correct output by more than 0.5, otherwise fitness assignment was the difference from the input just as before.

These tasks are somewhat similar to the sequence generation tasks introduced in Yamauchi and Beer (1994), one of the differences being that there, networks had to output a sequence without seeing it before; reinforcement was provided either only by fitness assignment (in which case the sequence remained the same all time) or using a reinforcement signal. Sequence recall tasks are also related to the work reported in Grüning (2006), where recurrent neural networks were first fed with a sequence of symbols, and then, upon presentation of a special symbol, had to output the whole sequence either exactly as given or in reverse order (In that work, networks were trained using a variant of backpropagation through time.).

Experiments are reported here using configuration DL1 and 30 runs for each task. Results are shown in Table 3, and were tested for significance using the Wilcoxon rank-sum test on a performance criterion that is based on the number of evaluations, but takes into account the final highest fitness in runs that fail to find a perfect solution. For

comparison with speciation selection, rank selection has also been used, where a roulette wheel is used for minimizing stochasticity and probability of being selected is proportional to fitness rank, with the probability of the best network being twice the average probability.

It can be seen that using speciation selection, one can find solutions for relaxed tasks up to 4 bit sometimes. In the 1–3 bit tasks, evolution is about 3× faster when using the relaxed output criterion. Those runs that do not converge on a perfect solution, nevertheless, get quite close to it.

It is also instructive to inspect the networks that solved the respective problems. For the strict task, evolved solutions used between one and two hidden neurons (mean 1.3) for the 1 bit task; between two and four hidden neurons (mean 2.7) for the 2 bit task; and between three and seven hidden neurons (mean 4.0) for the 3 bit task. When using the relaxed output requirements, evolved networks for the 1 bit task used either 0 or 1 hidden neuron (mean 0.3). Evolved networks for the 2 bit task used between one and four hidden neurons (mean 1.7); those for the 3 bit task used between two and five neurons (mean 3.0); those for the 4 bit task used between three and five neurons (mean 3.9).

This means that about one neuron less is used on average for the relaxed task. Indeed in experiments not shown here, solutions for the strict version of the 1 bit task could not be found when the insert node operation was disabled such that no hidden neurons could evolve. Although a single neuron with a recurrent connection is enough for memorizing one bit, it is apparently not enough for producing a strong enough output signal. Also, it is interesting to note the roughly linear increase, both in mean and minimum number of hidden neurons for both task versions for growing memory requirements. This reminds one of a kind of "shift register" solution to the problem, which also would scale linearly. On the other hand, a single neuron would be enough to store arbitrary amounts of information provided its state space is fine grained enough; however, this approach would require some additional neurons for encoding and decoding. Upon examination of evolved solution, no single dominant construction principle of the networks can be found; the evolutionary algorithm seems to take whatever happens to be useful.

Importantly, simple rank selection performs significantly worse than speciation selection on all sequence recall tasks except the relaxed one bit task, where simple selection experiments typically find solutions in the initial population, which is more diverse than the initial population used for speciation selection. This clearly demonstrates that the speciation method is indeed superior for tasks that need to evolve network topology. Surprisingly, simple selection performs even significantly worse on the relaxed output criterion than on the strict one for the 3 and 4 bit tasks. Why this is so is not entirely clear, but it seems reasonable to say that it is the memorization part of the task which is difficult for simple selection methods because these algorithms are inferior for evolving neural network topology.

Further experiments indicate that incremental evolution techniques are useful for memory evolution is well. For example, when evolution for the relaxed 4 bit task ran for 1,000 generations, no additional solutions were found, such that still only 23% of the runs converged. By contrast, when the solutions for the relaxed 3 bit task were evolved for 500 more generations using the 4 bit fitness function, 63% of the runs were successful, which is significantly better. No solutions to the 5 bit relaxed task were found using direct evolution for 1,000 generations. But when the evolved 4 bit solutions were used for initializing the population, two runs (7%) converged. These solutions, of course, had very long genomes due to many generations of insertion-based evolution. They were, therefore, compressed in another 500 generations using a fitness function which punishes genome length. Each solution was subjected to 15 runs; the 30 compressed solutions were then evolved for 500 generations to solve the relaxed 6 bit task. A total of 57% of these runs found solutions. It should be noted that not all bit combinations were tested for these tasks, but just eight randomly chosen ones. Still, these results show the power of this incremental neuroevolution method.

Conclusions

Benchmarking neuroevolution methods is important because when trying to find solutions for a new task, one wants to use a powerful method, and to be able to understand how difficult that task actually is for certain kinds of methods. The experiments on the pole balancing benchmark task revealed several important points:

- NEON can, despite some performance loss which may be due to its underlying indirect encoding, solve the DPNV and AWDPNV tasks
- Speciation selection significantly contributes to this success, although the gap to standard selection methods is not as big as reported in Stanley and Miikkulainen (2002)
- The structural developmental operators of NEON can indeed often avoid negative fitness effects, thereby contributing to the success of the method.

The latter two results are also relevant to the original NEAT method.

Sequence recall tasks are useful for benchmarking neuroevolution methods on the ability to find solutions for non-Markovian tasks, and build neural networks with hidden neurons. The experiments here showed that

- The NEON method can solve these tasks up to 4 bit
- Speciation selection is even more important on these tasks than on pole balancing because larger topologies are required
- Solutions for more than 4 bit of memory can be found using incremental evolution.

Again, these results are also relevant for the original NEAT.

Finally the experiments on evolution of complex genetic architectures show that NEON is able to find compressed genotypes through incremental evolution. It should, therefore, be feasible to store structure for new features uncompressed in the genome at first, and later reorganize and compress it as these features get conserved. This incremental approach can, in principle, overcome the problem of generating complex, not easily compressible phenotypes that plagues many developmental encodings.

Acknowledgments I would like to thank Jürgen Jost for support and helpful discussions.

References

Bentley P, Kumar S (1999) Three ways to grow designs: a comparison of three embryogenies for an evolutionary design problem. In: Proceedings of the genetic and evolutionary computation conference

Bongart J (2003) Incremental approaches to the combined evolution of a robot's body and brain. Ph.D. thesis, Universität Zürich

Eggenberger P (1996) Cell interactions as a control tool of developmental processes for evolutionary robotics. In: "From Animals to Animats 4", 4th international conference on simulation of adaptive behavior

Federici D (2005) A regenerating spiking neural network. Neural Netw 18:746–754

Green C (2006) SharpNEAT. http://www.sharpneat.sourceforge.net

Gruau F (1994) Neural network synthesis using cellular encoding and the genetic algorithm. Ph.D. thesis

Grüning A (2006) Stack-like and queue-like dynamics in recurrent neural networks. Connect Sci 18(1):23–42

Harding S, Miller J (2006) A comparison between developmental and direct encodings—an update of the GECCO 2006 paper "The Dead State". http://www.cs.mun.ca/~simonh/

Inden B (2007) Stepwise transition from direct encoding to artificial ontogeny in neuroevolution. In: Fernando Almeida e Costa et al. (eds) Proceedings of the European conference on artificial life

Nolfi S, Floreano D (2000) Evolutionary robotics—the biology, intelligence, and technology of self-organizing machines. MIT Press, Cambridge

Roggen D, Federici D (2004) Multi-cellular development: is there scalability and robustness to gain? In: Yao X et al (eds) Proceedings of the parallel problem solving from nature conference

Rylatt RM, Czarnecki CA (2000) Embedding connectionist autonomous agents in time: the 'Road Sign Problem'. Neural Process Lett 12:145–158

Stanley KO (2004) Efficient evolution of neural networks through complexification. Ph.D. thesis, Report AI-TR-04-314, University of Texas at Austin

Stanley KO, Miikkulainen R (2002) Evolving neural networks through augmenting topologies. Evol Comput 10(2):99–127

Stanley KO, Miikkulainen R (2003) A taxonomy for artificial embryogeny. Artif Life 9(2):93–130

Toussaint M (2003) The evolution of genetic representations and modular adaptations. Ph.D. thesis, Ruhr-Universität Bochum

Wieland AP (1991) Evolving controls for unstable systems. In: Touretzky D (ed) Connectionist models, proceedings of the 1990 summer school

Yamauchi B, Beer R (1994) Sequential behavior and learning in evolved dynamical neural networks. Adapt Behav 2(3):219–246

Ziemke T, Bergfeldt N, Buason G, Susi T, Svensson H (2004) Evolving cognitive scaffolding and environment adaptation: a new research direction for evolutionary robotics. Connect Sci 16(4):339–350