Evol. Intel. (2011) 4:145–163
DOI 10.1007/s12065-011-0057-0

RESEARCH PAPER

Evolving scalable and modular adaptive networks with Developmental Symbolic Encoding

Marcin Suchorzewski

Received: 1 July 2010 / Revised: 17 March 2011 / Accepted: 8 April 2011 / Published online: 3 May 2011

Abstract Evolutionary neural networks, or neuroevolution, appear to be a promising way to build versatile adaptive systems, combining evolution and learning. One of the most challenging problems of neuroevolution is finding a scalable and robust genetic representation, which would allow to effectively grow increasingly complex networks for increasingly complex tasks. In this paper we propose a novel developmental encoding for networks, featuring scalability, modularity, regularity and hierarchy. The encoding allows to represent structural regularities of networks and build them from encapsulated and possibly reused subnetworks. These capabilities are demonstrated on several test problems. In particular for parity and symmetry problems we evolve solutions, which are fully general with respect to the number of inputs. We also evolve scalable and modular weightless recurrent networks capable of autonomous learning in a simple generic classification task. The encoding is very flexible and we demonstrate this by evolving networks capable of learning via neuromodulation. Finally, we evolve modular solutions to the retina problem, for which another well known neuroevolution method—HyperNEAT—was previously shown to fail. The proposed encoding outperformed HyperNEAT and Cellular Encoding also in another experiment, in which certain connectivity patterns must be discovered between layers. Therefore we conclude the proposed encoding is an interesting and competitive approach to evolve networks.

M. Suchorzewski (✉)
Artificial Intelligence Laboratory,
West Pomeranian University of Technology,
ul. Żołnierska 49, 71-210 Szczecin, Poland
e-mail: msuchorzewski@wi.zut.edu.pl

Keywords Neuroevolution · Developmental system · Neural network · Genetic encoding

1 Introduction

As computers gain computing power, practical potential of evolutionary computation does grow as well. Evolutionary synthesis of intelligent agents, hardware or software, is a field of continuously growing interest. Agents can be evolved to solve virtually any reproducible problem for which a fitness function can be defined, and the most interesting results might be expected in domains, where little human expertise and no robust methods exist so far.

One promising approach to evolve intelligent agents is to combine learning and evolution in the evolutionary artificial neural networks (EANNs) or neuroevolution framework. Evolutionary algorithm (EA) can be used to optimize network topology, weights, transfer functions or learning rules. While plenty of EANN systems has been proposed so far [see e.g. 7, 33], most of them addressed only one or two selected aspects of network architecture. Less common are attempts to capture most of the architecture in the representation. And even more difficult is to find encodings designed with scalability in mind—property conducing evolution of complex, somehow regular networks.

The motivation behind using evolution to generate networks is probably well known to Evolutionary Intelligence journal reader. Nonetheless it might be worth brief recapitulation here. First of all, the same EA can be used across many different problem domains, while requiring little knowledge about them. The very flexible definition of the fitness criterion allows to generate networks having an arbitrary performance measure optimized—be it accuracy, efficiency, robustness or any combination of these.

Likewise, user-specified design constraints can be easily imposed, such as input-output interface or types and numbers of nodes and connections. EANNs can be optimized along a single or multiple dimensions, either implicitly or explicitly, by using an appropriate multi-objective EA [1]. However, there are also disadvantages and limitations of the evolutionary approach—primarily unbounded computing power demands and difficulties in the analysis of evolved solutions.

The main challenge in pursuit to evolve complex networks efficiently is their genetic representation. In this paper we propose a novel encoding for networks, called Developmental Symbolic Encoding (DSE). As the name suggests, the encoding is developmental, which means networks are grown according to some genetic recipe. The genome is a tree of routines, which in turn consist of lists of instructions. Genetic program grows the network by dividing nodes and layers and by connecting them in a more or less patterned manner. To this extent the encoding incorporates some concepts of two related neuroevolution methods—Cellular Encoding [9] and HyperNEAT [27, 30].

The encoding proposed allows to grow modular and regular networks in a scalable way. In a broad sense, *scalability* means a capability to solve varying size, and thus also large-scale, problems efficiently. This in turn implicates a capability to capture regularities inherent in these problems. Scalability manifests itself as a slower growth of the genotype as compared to the phenotype of network solution—a sign that some regularity of the problem has been reflected in the network and captured in its genotype. One of the most important elements influencing scalability is a capability to produce *modular* networks, i.e. networks consisting of structurally localized and functionally encapsulated subnetworks. From a topological perspective, a module is a set of nodes densely connected internally and sparsely connected to other nodes. Modularity is an important feature because it facilitates code reuse and exchange of useful modules between networks. That is why a significant amount of research has been devoted to modularity in evolutionary computation, as e.g. in Genetic Programming (GP) [see e.g. 22, ch. 6.1].

The two features of DSE, scalability and modularity, are demonstrated experimentally. In Sect. 4.1 we evolve perfectly scalable solutions to parity and symmetry problems. In Sect. 4.2 we demonstrate modular capabilities on the parity problem. In Sect. 4.3 we evolve modular networks capable to learn autonomously in a generic classification task and manifesting scalability in solving the task for increasing number of inputs.

Proposed encoding is very flexible in that it can generate whole array of networks—weighed and weightless, recurrent or feed-forward, employing arbitrary transfer functions as well as connections types. Depending on a variant of the evaluation algorithm, they can learn autonomously, through plastic connections with local learning rules, employ neuromodulation, and even backpropagation. It is possible to evolve learning rules for connections and transfer functions for nodes are also evolvable in principle; which is by itself an interesting subject for investigation [see 20]. In theory, the encoding can express any recurrent network, thus allowing to evolve networks equivalent to any Turing machine and solve any computable problem.

Yet, even with all these features, it remains difficult to estimate practical utility of the encoding. Due to its complexity, it has been tested on a few specific problems so far. It is also not straightforward to compare it with other neuroevolution methods. In Sect. 5, however, we manage to compare DSE with HyperNEAT and Cellular Encoding on two problems and the results show DSE is very competitive. Unlike its relatives, DSE succeeded in delivering modular solutions to the retina problem. It also outperformed the two other encodings in a task similar to the bit mirroring problem [3], in which regular patterns of connectivity must be discovered between two layers of nodes.

In Sect. 2 we briefly describe several notable approaches to evolve networks. In Sect. 3 we describe an adopted computational model of the network and the encoding itself. In Sect. 4 we examine the concepts of scalability and modularity of genetic representation and demonstrate these two features in DSE; we also demonstrate some flexibility of DSE with an example of network using neuromodulation to learn. Experiments are presented in Sect. 5 and finally we conclude the paper in Sect. 6.

2 Related work

An excellent review of the work on EANNs prior to 1999 can be found in [33]. More recent developments are covered in [6]. Therefore we briefly characterize only a few selected neuroevolution systems. For an overview of many other developmental systems, not necessarily devoted to evolve networks, see [12, 29].

2.1 Kitano's graph grammar

One of the earliest attempts to evolve neural networks, taking advantage of indirect encoding, was Kitano's Graph Grammar Encoding [16]. In this encoding, the neural network is represented with a genome divided into blocks of 5 elements, interpreted as fixed-length rewriting rules. The first element of the block represents a non-terminal predecessor (the left hand side of the rule), while the remaining four determine 2×2 matrix of terminals and non-terminals, being a successor (the right hand side). Starting with some initial non-terminal symbol, rules are

applied repetitively, building up the connectivity matrix until all its elements are terminals, i.e. 0's or 1's. The resulting network topology is then trained using error backpropagation algorithm. Kitano claimed the system allowed to generate better performing networks, however later it was suggested the difference in performance might be also due to other factors than genetic representation [see 33].

2.2 Cellular encoding

Another well known developmental representation for neural networks is Cellular Encoding (CE) [9], strongly inspired by the processes of biological development, mainly cell division and differentiation. Starting with an embryo—a single cell or neuron connected with all the inputs and outputs—the network grows according to a tree-structured program. Whenever the program tree branches, the cell divides in a way determined by the instruction in the branching node, with parallel and sequential divisions playing a major role. After division, daughter cells follow their own program-branches, and may undergo other transformations such as cutting and "weighting" incoming connections. CE was demonstrated to bear some important characteristics, such as completeness and closure. Moreover, if some recurrence is added to the program-tree, e.g. by means of conditional jumps (making it rather graph than tree), then CE can display modularity and scalability properties. CE was capable of growing networks solving 21-input parity problem, 40-input symmetry and 7-input decoder problem. Later it was used to generate locomotion controllers for multi-legged robots [10]. There is not much data, however, on CE's performance as compared to other neuroevolution methods. The only experiments we know are presented in [11], where it performs worse than a direct encoding in pole balancing tasks. Finally, CE was not designed to grow networks with regular patterns of connectivity between layers, and this is evident in the experiment presented in Sect. 5.1.

DSE is similar to CE in the way it grows the network, that is by means of parallel and sequential divisions of nodes, which alone can yield any number of nodes, deployed in fully connected layers. The way the network is connected, however, is very distinct. In CE, connections are manipulated using link registers, which are incremented, decremented and eventually point to the connection which can be cut. While this might be an efficient way of operating on the network, we believe it is brittle under genetic variation. Introduction of some new nodes to the network during variation is likely to disrupt the whole network. In fact, the work on DSE stemmed from our experiments with CE, where it was found difficult to grow networks capable of solving the perceptron problem even for $n = 1$ (Sect. 4.3). We suppose the difficulty might be partly due to that one fundamental trait of CE, that in the developing network, connections between neurons can be only cut and not established. Neurons can not establish new connections, because there is no way to select neurons beyond those already connected. Once the connection is cut, the set of possible network topologies achievable in further development is restricted. And if wrong cuts happen to occur in top parts of the developmental tree, which by chance dominates an early population, then it is—much as in the tree-based GP [19]—difficult to escape the local basin of topologies.

2.3 Analog genetic encoding

Another interesting system for neural development is Analog Genetic Encoding (AGE) [18], inspired by biological processes of gene regulation. Here the genotype is a string of characters from a finite alphabet, representing neurons ("devices") and weighted connections between them. The string contains predefined sequences of characters, which identify neurons of possibly various types. After the neuron identifier, a sequence encoding its weights follow, though the final weight values are computed from interactions between weight sequences of both neurons involved. AGE has been used to evolve several electronic circuits, such as temperature-sensing circuit or a circuit solving the XOR problem [18]. In [25] it was also used to evolve neuromodulatory network capable of learning in non-stationary environment—the foraging bee reinforcement learning problem. This, however, was achieved after biasing the system toward an adequate structure of the solution. The critical trait of AGE, in our view, is that the genotype encodes each neuron separately. This casts a doubt on its scalability, and in fact the issue is acknowledged by authors and considered for future work [18].

2.4 HyperNEAT

Scalability was in turn a major concern from the beginning in the work on Compositional Pattern Producing Networks and HyperNEAT [27], which seek to achieve the scalability offered by developmental systems, without simulating the process of development itself; or at least without striving to reflect biological development, which is based on local interactions between developing cells. Although the concept of local development is discarded here, the encoding is far from direct. In fact it employs another well known method for growing networks, namely Neuroevolution of Augmenting Topologies (NEAT) [28]. Networks grown by NEAT serve as genetic representation for the networks actually aimed to be evolved. Not going into details, the trick here is that although NEAT can not produce scalable

networks, it can produce networks, which can produce scalable networks. Using that approach authors interactively evolved 2D images having such properties as symmetry, imperfect symmetry, repetition and repetition with variation [27]. Then it was used to evolve controllers for food gathering simulated robots and networks solving a simple visual discrimination task at varying resolutions [30]. Since its conception, much further research on extensions and applications of HyperNEAT has been conducted [e.g. 8, 23, 24].

The basic idea behind HyperNEAT is very "neat" indeed: to encode the network topology using a single composed function; which in principle can be represented in many ways, and which composition determines the space of topologies easily expressible. Composing the function from symmetrical or cyclical subfunctions, for example, would yield topologies featuring some symmetry or repetition. Originally, HyperNEAT employs the so called Compositional Pattern Producing Networks (CPPN) to represent the function, but we have experimented with expression trees (borrowed from GP) also, producing general solutions to the symmetry and parity problems. This neat idea has been also introduced in DSE through ConE instruction, in which one of the arguments defines the connectivity between two groups of nodes (layers) by means of an expression tree. Using appropriate non-terminal nodes, provided the unlimited depth of the tree and unique indexes of nodes, it can describe any connectivity between two layers, or within a single layer. So the power of expressiveness is ultimately the same as in CPPNs, although they work a bit differently. First, CPPNs operate on the basis of nodes' geometric positions in n-dimensional space called substrate, whereas ConE instruction operates on indexes (which can be also multidimensional in principle). Second, CPPNs compute the connectivity for the entire network at once, while ConE can perform calculations for two selected groups of nodes only. It means growing a multi-layer network with ConE instructions can be expected to be computationally less expensive than with CPPN.

One severe limitation of the original version of HyperNEAT was the necessity to determine the number and geometric placement of nodes in advance. Only recently, in [24], some way to include the number and placement of nodes in the encoding has been proposed. Yet, the mechanism is complicated and has been demonstrated on a single test problem so far, namely navigation in T-maze. In DSE, the number of nodes and their labeling is controlled by Div* and Subst instructions, so the size of the network can freely evolve and match the problem without presumptions. Still, it is possible to constraint the size of the network, by introducing explicit limits or by disabling Div* instructions.

Another important difference is the evaluation mode of network phenotypes. In HyperNEAT, the network is always evaluated in "virtual parallel" manner. This makes evaluation of feed-forward networks very expensive. Eventually, some execution order could be assumed in the network, again however, that would be an assumption. In contrast, nodes are explicitly ordered in DSE and that order can evolve; and it only requires a minor modification in the evaluation algorithm (see Sect. 3.1) to obtain a virtual parallel mode of execution. So DSE appears to be again more flexible.

Finally, an important question arises, whether HyperNEAT supports modularity. From one side, it is capable of producing many repeated subnetworks with a single piece of genotype, but from the other side, that piece of code is usually inseparable from the whole genotype. The genotype is inherently non-modular in the sense, that no fragment of it can be extracted, which would alone produce some particular subnetwork. As a consequence, individuals can not exchange "recipes" for useful subnetworks, as in case of DSE solving the perceptron problem (Sect. 4.3), where the solution for $n = 1$ instance was used to solve $n > 1$ problem instances. Noteworthy is that in [31] it has been shown, that it is possible for HyperNEAT to solve one problem more effectively if there is already a solution for a simpler version of that problem; this demonstrated some versatility of HyperNEAT solutions, yet that is a different concept than modularity and transferability of modules.

2.5 Summary

In the end, an important question is what new DSE has to offer. As it was already remarked, DSE combines some concepts of CE and HyperNEAT. Much as CE, it can grow networks by means of node divisions, and features an explicit genetic modularity and hierarchy, conducing reuse of code and network modules. Much as HyperNEAT it can establish connectivity patterns between layers or groups of nodes, and exploit some geometric-like relationships between them, enabling evolution of highly regular network topologies. These two ways of growing networks are combined in a coherent genetic representation, optimistically allowing to get best of both encodings while solving problems.

3 Developmental Symbolic Encoding

DSE defines genetic representation for networks along with some genetic operators working on it. As such, it forms the central part of EA, which is nevertheless quite independent from other parts, such as fitness evaluation, selection or population structure. In this section, therefore, we focus on

the encoding itself, assuming the rest of EA is typical, as for example in Genetic Programming (GP) [22]. We believe the basic results presented in Sect. 4 should also be reproducible under different settings of EA.

3.1 Phenotype

The phenotype of the individual is the network $G = (\mathbf{V}, E) = ([v_1 \ldots v_N], \{e_{ij}\})$, where $\mathbf{V}$ is a vector of nodes and E is a set of connections. Nodes and connections can hold a number of attribute values. Some of them playing role in the development, while other being used only in computation. Throughout the paper we assume nodes have only two attributes—an indexed label (*symbol*) and the current activation value. Connections, in turn, can have disparate number of attributes, depending on the type of connection. Weightless connections does not need any attributes, weighted connections need only a weight, while plastic connections can hold a number of parameter-attributes. Such a representation of G is a bit different from a conventional notation for directed graphs, where nodes and connections are contained in sets, and attributes are eventually imposed using functions. This departure follows from the assumed sequential model of computation, in which nodes are evaluated in an ordered manner. Therefore it is natural to have them explicitly ordered in a vector.

Vector $\mathbf{V}$ can be dissected into three contiguous subvectors: input $\mathbf{V}_u$, hidden $\mathbf{V}_h$ and output $\mathbf{V}_y$. Input and output subvectors constitute the interface of the network and their length depends on the problem. The hidden part of the network is free to vary in size.

The network is initialized only once, at the beginning of its "lifetime" or a trial. During initialization, values of all nodes are zeroed and weights are set to their initial values. Then, for any input vector $\mathbf{u}$, the network is evaluated as follows:

1: Assign inputs $\mathbf{V}_u := \mathbf{u}$
2: **For** each $v_j \in [\mathbf{V}_h \mathbf{V}_y]$ **do**
3: Evaluate node: $x_j := f_j(\{e_{ij}(x_i)\}_{i=1,\ldots,n_i})$
4: **For** each connection e_{ij} **do**
5: Update e_{ij} (if applicable)
6: Return $\mathbf{V}_y$,

where x_j is the node's value, $e_{ij}(x_i)$ denotes an operation performed by the connection—usually just weighting, and f_j denotes a transfer function assigned to node j. The transfer function operates on the set of incoming signals, which are usually aggregated by summing. Note the formula for a connection update is not given, because it is dependent on the connection type; besides, it is only applicable in networks with plastic connections.

3.2 Genotype

The genotype is a kind of program to grow the network. The program is tree-structured, with nodes corresponding to routines. The root of the tree constitutes the main routine and it is sufficient to solve many test problems. Each routine-node of the program-tree has the same structure: $\gamma = (R, \text{Body}, \text{Tail}, \gamma_\gamma)$, where R is an identifier, Body is a list of instructions, Tail is a list of terminating instructions and γ_γ is a set of subroutines.

The main component of the routine is Body. It contains instructions acting on, and developing the network. The whole program Γ grows the network from its initial state into final, i.e. $G_\tau = \Gamma(G_0)$. So the instruction can be seen as a function λ taking the network G_t and returning a new network G_{t+1}, i.e. $G_{t+1} = \lambda(G_t)$. Let us briefly describe the most important instructions:

1. Con *X Y C*: connect nodes *X* and *Y*, with the connection having the same attribute values as a reference connection *C*.
2. Cut *X Y*: cut connections between *X* and *Y*.
3. ConE *X Y C E*: connect nodes *X* and *Y*, satisfying the expression *E*, using *C* as a reference connection.
4. CutE *X Y E*: cut connections between *X* and *Y*, satisfying the expression *E*.
5. DivP *X Y*: divide nodes *X* "in parallel". Parallel division duplicates the node *X* along with all its connections, assigns label *Y* to the new node and places it in $\mathbf{V}$ right after the original.
6. DivS *X Y*: as above, except the division is sequential, which means that instead of duplicating connections, *Y* takes over the outgoing connections of *X* and the connection from *X* to *Y* is set.
7. Subst *X Y*: substitute node symbols *X* with *Y*.
8. Call *R X*: for each node *X* call the subroutine *R*.
9. Term *X Y*: like Subst, except *Y* is necessarily terminal, i.e. it denotes a transfer function.

Here, the instructions' arguments *X* and *Y* stand for symbols, which are used to identify nodes in the network or elements in vector $\mathbf{V}$. Actually, the symbol *X* consists of label x and index i. Since many nodes can bear the same label, it serves to identify a layer or a group of nodes. The index is a natural number, but it can also take an "empty" value (ε). In that case, plain x means "all the nodes with label x". It makes possible to *select* and operate on a whole group of nodes with a single instruction. More formally, symbol x_i matches y_j if and only if: $x = y \wedge (i = j \vee i = \varepsilon)$.

The expression E appearing in ConE and CutE instructions is a tree involving some basic arithmetic and Boolean operations, constants and usually four inputs. The expression is evaluated for all node pairs from the two layers involved. Among inputs to the tree are indexes of nodes and the highest indexes in both layers. In effect, the connection of type C is established for all the node pairs satisfying the expression. See Sect. 4 for examples.

3.2.1 Initialization and determination

The initial population of genotypes consists of either empty or randomly generated genotypes. A random genotype consists of randomly drawn instructions, having their arguments *undetermined*. The way arguments are determined is an important part of the encoding, affecting its performance. Arguments of instructions have to correspond to the current state of the network, otherwise instructions are unlikely to be effective. Argument X in instruction DivP, for instance, has to match some nodes in the network or the instruction will be ineffective. Therefore, arguments can not be drawn in a completely random manner, but from certain „sources". To clarify it further, let us discern several classes of symbols:

1. Input, $S_u \sim \mathbf{V}_u \subset \{x_i : x = \mathsf{u}\}$.
2. Hidden, $S_h \sim \mathbf{V}_h \subset \{x_i : x \in \{\mathsf{A}, \ldots, \mathsf{Z}\}\} \cup S_t$.
3. New, $S_n \sim \{x_i : x \in \{\mathsf{A}, \ldots, \mathsf{Z}\} \wedge i = \varepsilon\}$.
4. Output, $S_y \sim \mathbf{V}_y \subset \{x_i : x = \mathsf{y}\}$.
5. Terminal, $S_t \sim \{x_i : x \in \{+, -, *, \ldots\}\}$.
6. Subroutine identifier, $S_r \sim \gamma_\gamma \subset \{x_i : x = \mathsf{r}\}$.

The sentence $S_a \sim X \subset Y$ means "the symbol of class a is *determined* by drawing from X, which belongs to Y domain". Such an interpretation pertains to input, hidden, output and subroutine identifiers (where γ_γ denotes the set of all subroutines' identifiers in a given routine). In case of new and terminal symbols, determination is done straightaway on their domain. Note that terminal symbols are those denoting some transfer function, all other symbols, except S_r, can be treated as non-terminal. The classification serves to clearly define valid arguments for instructions (Table 1). We write S_{ab} or S_{alb} to denote the argument can belong to class S_a or S_b.

Whenever a new instruction is inserted into the program, either during initialization or mutation, its argument symbols are undetermined; and they become determined during development. Determination of the symbol is tightly coupled with its class and consists in drawing the symbol from its "source" or domain. For S_{ab} the symbol is drawn uniformly from the union of S_a and S_b. The first argument of the Con instruction, for instance, is determined by uniformly drawing the symbol from input and hidden subvectors of $\mathbf{V}$. In case of S_{alb}, first a single domain is drawn with equal probability, and then the symbol from it (uniformly). These two ways of determination has been introduced to maintain a balance between the new symbols and those already existing in $\mathbf{V}$. Once the instruction is determined, it remains so, until eventually one of its arguments is mutated into undetermined form; and determined again during the next development.

The last note on argument determination regards „index stripping". Each time a node-selecting symbol is drawn (S_{uhy}), it is stripped from its index with some probability (0.5 by default); and in case of ConE and CutE instructions it is stripped every time. This is to allow an instruction to generalize and operate on a whole group of nodes instead of just a single one.

Table 1 Instructions and their valid arguments

Con S_{uh} S_{hy} C	Cut S_{uh} S_{hy}	Div* S_{uhy} S_{hln}	Subst S_{hy} S_{hlnlt}
ConE S_{uh} S_{hy} C E	CutE S_{uh} S_{hy} E	Call S_r S_h	Term S_{hy} S_t

3.3 Development

Much as in Cellular Encoding [9], development usually starts from an initial network G_0, having a single hidden node fully and weightlessly connected with inputs and outputs. The hidden node's initial symbol is always A_0. Alternatively, development can start from an empty network, consisting of inputs and outputs only. The final network G_τ is generated by executing the program in the genotype, i.e. $G_\tau = \Gamma(G_0)$.

Network G_τ is guaranteed to be functionally valid due to the Tail part of routine and the covering operator, which automatically inserts Term instruction for any non-terminal symbol in $\mathbf{V}$ (see Sect. 3.4). In fact, development can be interrupted at any instruction-step in the Body, and the network will be valid, provided it is terminated by the Tail.

Figure 1 gives a simple example of network development from its initial to the final stage. Nodes without inputs and having the transfer function $*$, produce constant 1 on output. The network computes a logical equivalence function or can act as a XNOR gate (assuming the output greater than 0.5 means 1 and 0 otherwise).

3.3.1 Subroutine development

An important feature of the encoding is its capability to reuse code be means of routines. When a subroutine R is called via Call R X, development proceeds as follows:

1: A new initial network $\hat{G}_0$ is created, in which the number of inputs corresponds to the number of connections incoming to X node, the number of outputs corresponds to the number of connections outgoing from X, and there is one hidden node A_0.

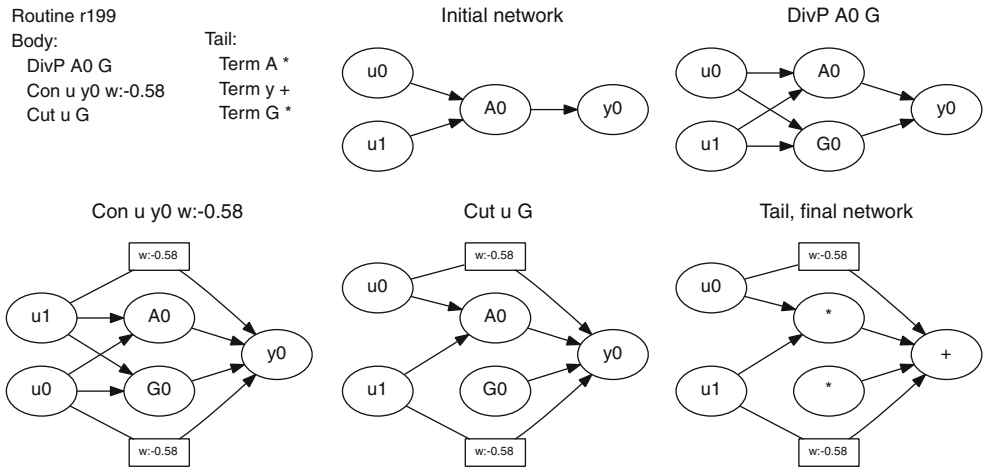


Fig. 1 Example of network development; given the genotype (*top left*) and the initial network (*top middle*), development proceeds with 3 Body and 3 Tail instructions

2: Subroutine $\hat{\gamma}$ is executed, i.e. $\hat{G}_\tau = \hat{\gamma}(\hat{G}_0)$.
3: The resulting subnetwork $\hat{G}_\tau$ is composed with theparent network G.

The composition of networks consists in replacing all input nodes in $\hat{G}_\tau$ with their corresponding nodes in G and connecting (with weights 1) all output nodes in $\hat{G}_\tau$ with their corresponding nodes in G. In short, node X is replaced with the subnetwork $\hat{G}_\tau$. Certainly, the symbol X may select several nodes in the network; then the subroutine is called for each one in sequence.

One critical aspect of subroutine development is how to map the nodes connected to X to the inputs of the initial subnetwork $\hat{G}_0$, and likewise the output nodes of X to the outputs of the subnetwork. Fully universal subroutines would require any such a mapping to be possible to express. One solution to that problem would be to add an additional attribute to the connection, allowing to order the connections (and thus the nodes in question) in arbitrary way. For the sake of simplicity however, we restrict the mapping to the ordering in vector $\mathbf{V}$, thus inputs (and outputs) in the subnetwork are always indexed according to the order of the corresponding nodes in $\mathbf{V}$ of the parent network G.

3.4 Genetic operators

We employ four operators that can modify the genotype Γ : mutation, crossover, cleaning and covering. Here we describe them in general terms, because details are often a matter of arbitrary choice and their impact on the performance is difficult to assess.

The most prominent is mutation. When an individual is selected for reproduction, mutation is performed with some fixed probability. The operator is defined to act on the program-tree in a recursive manner, i.e. it starts with the main routine, but eventually recurse deeper into the tree. We suppose, however, that for the evolution of subroutines—and thus modularity—it might be better to mutate subroutines less often. Essentially, the operator can insert a new (undetermined) instruction, delete one, mutate one, i.e. turn one of its arguments into undetermined form, or duplicate and mutate simultaneously.

Crossover operator deals with whole routines and is very similar to the crossover in tree-based GP [22]. In brief, the crossover replaces a randomly chosen subtree in the acceptor tree with a randomly chosen subtree from the donor tree. The choice of subtrees is random, yet constrained by the maximum depth of the tree and the maximum arity of the node in the tree, which can be imposed by the user.

Cleaning is not very essential operator, though it improves readability of genotypes and speeds up the search in some problems. It consists in removing all the ineffective instructions from the genotype. The instruction is ineffective if it does not alter the network in any way.

Unlike 3 previous operators, covering is not applied during reproduction, but at the end of development. As already mentioned, for each non-terminal symbol X remaining in $\mathbf{V}$ by the end of Tail, the covering operator appends a new Term X Y instruction. Covering may be applied for each routine or for the main routine only. Either way it guarantees, that all nodes in the final network are assigned valid transfer functions.

4 Properties of DSE

4.1 Scalability

What exactly is scalability? In the context of evolutionary networks, Harding and Banzhaf [12] say scalability

"means that it is possible to grow more neurons if they are needed in the neural net to solve harder problems". Hornby [14] and Lipson [17] decompose scalability into regularity, modularity and hierarchy. Stanley et al. [8, 30] view it as a capability to grow large-scale networks; to represent networks at any resolution, allowing them to scale to new numbers of inputs and outputs, possibly without further evolution. Gruau [9] formulated a strict—and very demanding—definition of scalability, according to which, it is the capability to encode the network solution for a problem with varying number of inputs, using a single (possibly parametrized) genotype. Closely related to this definition is the notion of compactness, which is the ability to encode the network compactly, hence to capture some of its regularity. Compactness manifests itself as a slower growth of the genotype than the phenotype.

In our words, scalability is a capability to reflect some regularity of a problem in the network and capture that regularity in the genetic code, and thus to solve increasingly larger problems more efficiently. Certainly, different problems can have different type and amount of regularity to be captured and thus different amenability to the scalable method.

One measurable symptom of scalability is a slower growth of the genotype as compared to the phenotype of network solution, which can be measured as the ratio:

$$S(n) = \frac{|\Gamma(n)|}{|G(n)|}, \quad n = 1, 2, \ldots, \tag{1}$$

where n is the size of problem, e.g. measured with the number of inputs; $|\Gamma(n)|$ and $|G(n)|$ denote sizes of the genotype and the phenotype, correspondingly.

The ultimate way to measure scalability, however, is to see how a computational effort needed to solve a problem grows with its scale. For perfectly scalable solutions, as conceived by Gruau's definition for example, the effort would be constant with problem size, i.e. $S(n) \in O(1)$.

In the remaining part of this section, we demonstrate on two examples, that DSE features scalability, i.e. it is capable to capture regularity of a problem and encode the network solution in a compact way.

4.1.1 Scalable solution to the symmetry problem

The problem of symmetry is to find a network computing the symmetry function of n binary inputs, i.e. returning 1 if $u_i = u_{n-i-1}$, $i = 0, \ldots, n-1$ and 0 otherwise, where $\mathbf{u}$ is the input vector.

We employ an incremental evolution approach. Initially the population solves $n = 2$ problem instance. When the solution is found, a new instance with $n = 3$ is added for evaluation, and so on, until eventually a general solution for all $n = 2, \ldots, 11$ is found, which is likely to work also for $n > 11$.

In Fig. 2 we show one such evolved solution to the symmetry problem. The network employs n-ary EQ (=) and AND (&) transfer functions, defined as:

$$EQ(\mathbf{x}) = \begin{cases} 0 & \text{if } |\mathbf{x}| = 0, \\ x_0 & \text{if } |\mathbf{x}| = 1, \\ 1 & \text{if } |\mathbf{x}| > 1 \wedge \forall_{i \neq j}\, x_i = x_j, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

$$AND(\mathbf{x}) = \begin{cases} 0 & \text{if } \exists_i\, x_i = 0, \\ 1 & \text{otherwise,} \end{cases}$$

where $|\mathbf{x}|$ denotes the number of elements in vector. The genotype encodes the network that automatically scales up to the problem dimensionality during development. This is done in three constructive steps: first, the input layer is divided sequentially, producing nodes 10–14, second, another division produces nodes 5–9, and third, connections are made between the input layer and layer B, according to the tree expression (in prefix notation) in the ConE instruction: $i + j = m + 0$, where i and j are node indexes in both layers and m is the maximum index in the target layer (B). So the connection of type '1' (weightless) is made for every pair of nodes from both layers satisfying the expression. Finally, transfer functions are assigned by the Tail. The solution shown is the simplest among several other obtained, though it is obviously not minimal. Hidden nodes are numbered according to their position in $\mathbf{V}$, so it might be noted the network is feed-forward.

4.1.2 Scalable solution to the parity problem

Parity is another well known machine learning test problem. Here the objective is to calculate for each of 2^n possible input vectors $\mathbf{u}$, whether it contains even number of 1's. Figure 3 depicts one evolved solution to this problem, where the only transfer function used is NEQ, defined as a negation of EQ.

Capability to produce perfectly scalable (i.e. general) solutions to the parity problem has been also recently demonstrated for Self-modifying Cartesian Genetic Programming [13].

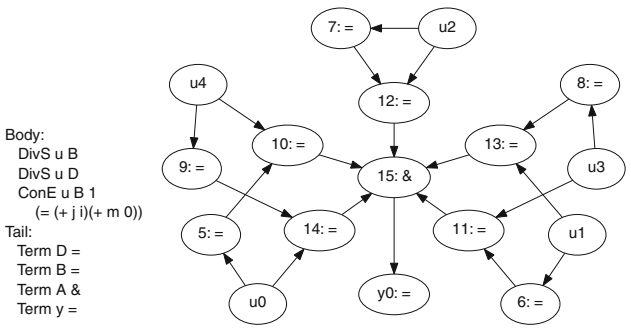


Fig. 2 Perfectly scalable solution to the symmetry problem; the network shown is developed for $n = 5$ inputs

4.2 Modularity

From a general network theory perspective, the network is modular, if it contains subnetworks densely connected internally, but sparsely connected between themselves. Kashtan and Alon [15] define modularity as "the separability of the design into units that perform independently, at least to a first approximation", which is then quantified by a measure coming from the general network theory proposed by Newman and Girvan [21]. Clune et al. [5] follow these lines by defining modularity as "the localization of function within an encapsulated unit, which in a network entails clusters of nodes with high connectivity within the cluster and low connectivity to nodes outside the cluster". This is also close to the view of "modularity as an encapsulated group of elements that can be manipulated as a unit" by Hornby [14], where "manipulated" presumably includes "reused". Perhaps the shortest formulation is by Lipson [17], according to which "functional modularity is a structural localization of function".

Another definition given by Gruau [9] and referring specifically to genotypic modularity, says that if network G_1 contains many copies of subnetwork G_2 and the genotype of G_1 includes the code for G_2 only once, then the genotype for G_1 is modular. This definition, however, does not cover networks with single instances of modules, which can possibly be replaceable and transferable between networks.

Bringing these views together, we say genetic encoding supports modularity, if the genotype is capable of producing "encapsulated" and possibly repeated subnetworks in the network using a single piece of code. Modularity can be further enhanced by the capability to transfer genetic code producing phenotypic modules between individuals. This would be particularly interesting in scenarios, where many related problems are solved in parallel and hence benefits from the communication might be expected.

Closely related to modularity is hierarchy, which is conceived as "the recursive composition of structure and/or function" [17]. Support for hierarchy by the encoding makes it theoretically possible to decompose the network into smaller, possibly nested and reused subnetworks and eventually to solve complex problems more efficiently.

Modularity and hierarchy are explicitly supported by DSE, as the genotype is a tree of routines, here. These routines can produce multiple subnetworks in the network solution. Moreover, they are easily transferable between individuals, making it possible to take advantage of parallel task solving. In the following, we demonstrate modular capability of DSE on example.

4.2.1 Scalable and modular solution to the parity problem

Using the same incremental approach as in previous examples, we evolve a modular solution to the parity problem (Fig. 4). This time we employ a different set of transfer functions: NAND (ṅ), OR (|) and AND (&). Here, NAND is defined as a negation of AND (Eq. 2) and OR returns 1 only if there is an input having value 1, and 0 otherwise. The solution is partially scalable—it works for $n = 2, \ldots, 9$, but fails for higher number of inputs. Routine r_{214} is used to produce the chain of subnetworks computing 2-input NEQ (XOR) function. The network shown contains 3 such subnetworks, each tied to 1 input (nodes 6–9 and u_0, 10–13 and u_1, 14–17 and u_2). Note the working of the routine is a bit tricky, as routines can behave differently depending on the number of input connections to the node for which they are called.

The example also illustrates two aspects of scalability. First, the solution is (partially) scalable in the strict sense, where a single genotype encodes solution networks for several different problem sizes; and second, the genotype is scalable in the broader sense, because it captures some regularity of the problem and narrows the search—as might be noticed, the effort of finding solutions for $n > 9$ is reduced to adding successive calls to the subroutine.

4.3 Scalability and modularity in a supervised learning problem

An important focal point in the work on the encoding was the capability of networks to learn, and especially to learn autonomously, i.e. by means of their internal dynamics and not by some externally crafted algorithm. Is it possible for the evolution to discover a genuine way of learning for networks, resulting from the network structure and

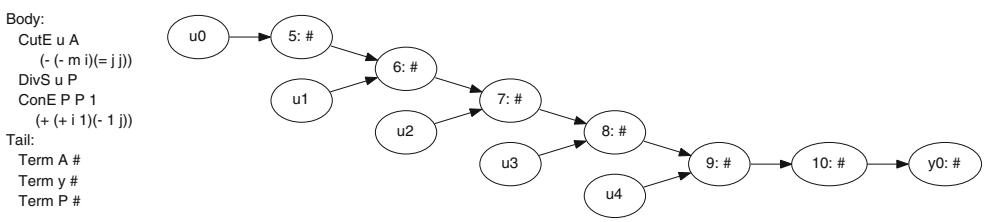


Fig. 3 Perfectly scalable solution to the parity problem; the network shown is developed for $n = 5$ inputs

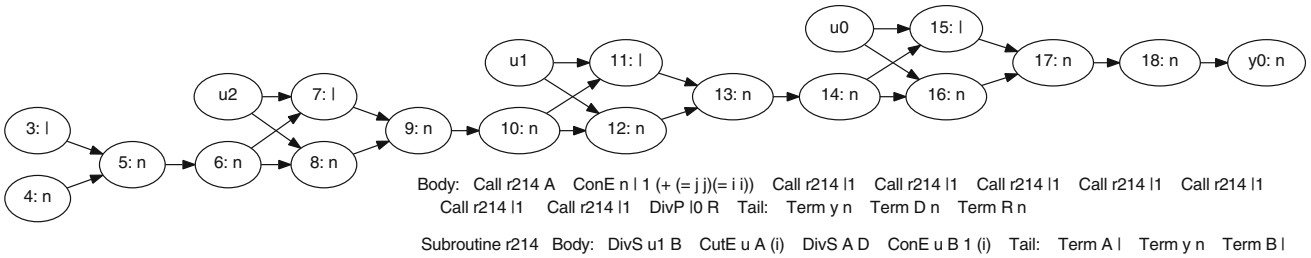


Fig. 4 Modular, partially scalable solution to the parity problem; the network is developed for $n = 3$ inputs

dynamics, instead of some predefined learning rules? Originating from that question is the problem of emulating a perceptron neuron capable of learning linear discrimination, by means of primitive transfer functions only ($+$, $\star$) and weightless connections. We call it "perceptron problem" in short.

Let $\mathbf{x} \in \mathbb{R}^n$ be a point (vector) drawn from a multivariate normal distribution with mean 0 and standard deviation 1 in all dimensions. Let $X = \{\mathbf{x}_i;\ i = 1, \ldots, 20\}$ denote training set of 20 points. Now, let draw another n-dimensional vector $\mathbf{w}$ from likewise multivariate normal distribution $N(\mathbf{0}, \boldsymbol{I})$, which will act as a hyperplane bisecting the space into two classes. For each vector $\mathbf{x}_i$ calculate its class using scalar product as follows:

$$y_i = (\mathbf{w} \cdot \mathbf{x}_i > 0), \tag{3}$$

assigning 0 to points lying on one side of hyperplane and 1 to those on the other. Next, we draw a translation vector $\mathbf{z}$, from $N(\mathbf{0}, \mathbf{I})$ again, and translate all the points from X by that vector. As a result we obtain a set of 20 randomly labeled, yet linearly separable points, distributed normally around some point near the origin of a coordinate system. Then a candidate solution is evaluated as in Algorithm 1.

The algorithm iterates over 10 trials. From each trial it calculates the average number of misclassifications, but taken from the last epoch only. If the program classifies all inputs correctly in some epoch, the trial is interrupted with perfect score 0. In each generation, the best individual in the population is tested not for 10, but 300 trials. Note we do not perform any validation within trial, because we are only interested in basic learning capability, so it is sufficient to observe learning on the training set only. To avoid bias in reported results, however, the best individual in the run is tested once more, and the result is taken as the final performance result.

In the experiment, we use incremental evolution approach, though a bit different than in previous examples. The evolutionary run starts with a single population (of size 2,048) solving the problem for $n = 1$ data inputs (and the error input). Whenever solution for n-input case is found, a new population for $n + 1$ instance is created. In order to save some computation, all pre-existing populations are halved. In each generation, each population exchanges 1% of its individuals with another randomly chosen population.

Due to randomness in data and time limits on training, it is difficult for a network to obtain a perfect score in the test evaluation. Even perceptron neurons trained with a backpropagation algorithm fail to learn occasionally. As a working condition, we consider the network to be a solution if $J < 0.1$, i.e. if it classifies correctly more than 90% of data points on average. Table 2 compares results from 100 trials with a standard perceptron (Neural Network Toolbox for Matlab) and from 20 evolutionary runs on this task. Note, the figure for $n = 6$ is less meaningful, as the run was stopped as soon as the solution for this task was found, so the population had no time to further improve in this case.

In all 20 runs, solutions for all task cases were found in 8,000 generations. In Table 2 we also show how difficult it was to find a solution for each problem instance. Interestingly, the most effort is required to discover the solution for $n = 2$, given the solution for $n = 1$ has been found. Although finding the solution to the $n = 1$ instance is also difficult. However, the effort does not seem to grow further with dimensionality.

The performance of evolved networks is similar to the generic perceptron neuron trained via error correction rule. Noteworthy, for lower dimensions they can even outperform the generic network. In a separate experiment, we compared the performance of the best evolved network for $n = 1$ to the perceptron network. In 2,000 trials, the evolved network learned the classification perfectly for all training sets, with a mean number of learning epochs about 2.3. The generic perceptron failed to learn to classify perfectly about 14% of training sets, and the mean number of epochs was about 3.3. Also noteworthy was the time required to complete the task: about 400 s in case of the perceptron implementation and 0.07 s in case of the network written as an array of equations. Figure 5 presents one such a solution.

Figure 6 (left) shows how the growth in the genotype translates into growth in the network. Data points represent average values from all solutions from each run and each problem instance. As can be seen, a logarithmic curve can be well fitted here, indicating the genotype grows much slower than phenotype. Another fit in Fig. 6 (right), shows

Algorithm 1 Evaluation of a candidate solution

```
Let criterion J := 0
For 10 trials do
    Recreate the task, i.e. training sets X, Y
    For8 epochs do
        Permute points in X and labels in Y correspondingly
        Set error e := 0 and cumulative error e_cum := 0
        For each point x_i ∈ X do
            Calculate the output of the network y := φ([e x_i])
            Calculate the error e := y_i − (y > 0.5), which is either −1, 0, or 1
            e_cum := e_cum + |e|
        if e_cum = 0 then
            Proceed to the next trial
J := J + e_cum/(10 ∗ 8)
```

Table 2 Best and mean performance in terms of misclassification percentage of a standard perceptron and best evolved networks; also the mean number of generations elapsed between finding solutions for $n-1$ and n problem instances is given

		n					
		1	2	3	4	5	6
Perceptron	Min	1.0	2.4	3.7	4.1	4.4	4.5
	Mean	1.3	3.3	4.5	5.2	5.3	5.3
	Std	0.2	0.5	0.6	0.5	0.5	0.5
Evolved	Min	0.0	1.2	2.6	4.7	5.1	(5.4)
	Mean	1.2	4.0	5.0	6.0	7.1	(9.8)
	Std	0.7	1.0	1.2	1.3	1.6	(2.3)
No. generations	Mean	264	777	126	77	170	106
	Std	346	1,627	82	82	321	95

the number of developmental steps versus the network size, fitting a power curve this time. The figure allows to better understand where does the scalability comes from in DSE—it is the result of two main factors: routines and symbol multiplicity (when a single instruction operates on several nodes in **V**). This is because exploitation of routines tend to prolong development, while having little effect on a genotype size. Symbol multiplicity in turn, affects neither development time nor genotype size. Therefore, because the growth of genotype (logarithmic) is slower than development time (square root), which is in turn slower than linear, it might be supposed, that scalability is due to both factors.

Also noteworthy is that we performed another experiment, in which subroutines were disabled. The results were much worse, in that solutions for $n > 1$ were not found. Therefore we conclude the success of DSE on this task is much due to discovery and effective use of routine modules, including effective communication between populations solving different problem instances.

Figure 7 presents an example of evolved network and its genotype for $n = 4$ data inputs (u_0 is the error). The striking feature of the network are its repeated subnetworks for each data input, appropriately tied together by the error input. The additional "inputless" subnetwork acts as a bias (inputless * nodes produce 1 on output). What is also important, all subnetworks are evidently generated by the single routine r_{159}, which is executed six times during development (1 call for 6 A nodes in **V**). Let us shortly describe how the genotype produces the network. First, node u_3 divides in parallel, making a seed for the "bias" subnetwork. Next, the input layer divides sequentially, making 6 A nodes in total (5 + 1 initial). Finally, after some re-connections, the subroutine is called on these A nodes, producing the subnetworks.

Interestingly, the common identifier of the subroutine and the main routine (also r_{159}, though not shown) suggests, that they evolved from a common ancestor routine. Note also the 3rd instruction in the main routine, indicating it came from a population solving an instance with $n > 4$.

A brief analysis of the functionality of the network reveals that each subnetwork performs a computation, which corresponds to weight multiplication and update with delta rule in standard perceptron. If we call the node with recurrent connection (e.g. node 14) an analog of weight, w, then the output v of each subnetwork associated with data input u (e.g. 16) is computed as:

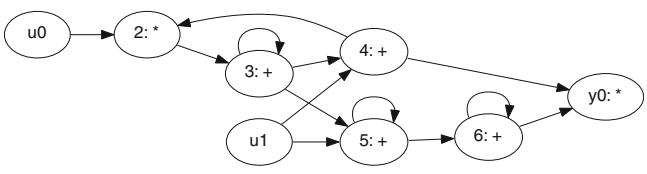
$$v(t) = u(t)w(t) \quad ; \quad w(t) = w(t-1) + \eta u(t-1)u_0(t). \tag{4}$$

4.4 Flexibility

Beside examples from previous sections, we applied DSE to several other test problems. For the artificial ant on Santa Fe trail—well know problem in GP community—we evolved two kinds of network solutions: weighted networks using + as the only transfer function, and weightless

Fig. 5 Network outperforming standard perceptron in the classification task for $n = 1$ data inputs

```
Body:
  DivS u0 P
  DivP A E
  DivS E D
  Con E E0 1
  DivS u0 C
  Con D D 1
  Con A0 C0 1
  Con P0 P0 1
Tail:
  Term A +
  Term P +
  Term E +
  Term D +
  Term y *
  Term C *
```



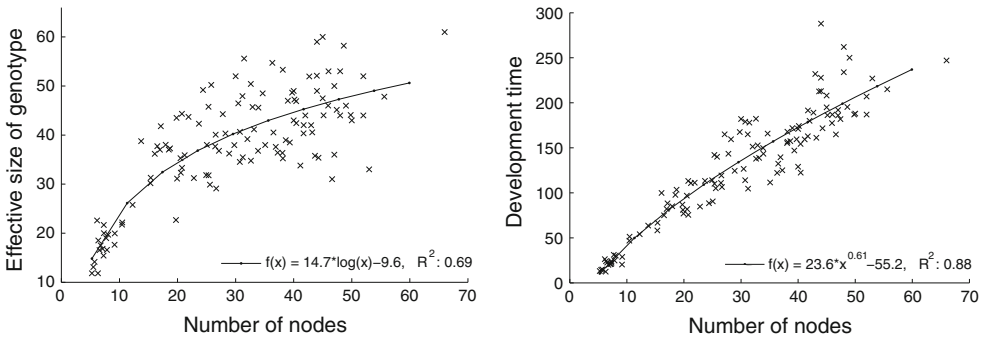


Fig. 6 A logarithmic fit on genotype vs phenotype size data (*left*) and a power fit on development time vs phenotype size data (*right*)

networks using $+$, $-$, $*$, h and m as transfer functions, where $h(\mathbf{x}) = ((\sum \mathbf{x}) > 0.5)$ and $m(\mathbf{x}) = (\sum \mathbf{x})/|\mathbf{x}|$.

DSE was also successful in evolving controllers for double pole balancing task (i.e. two inverted pendulums mounted on a cart side by side), with and without velocities. As this problem is rather irregular, DSE's performance could not possibly match the performance of NEAT [28].

In a similar task to the perceptron problem (Sect. 4.3), we also tried to evolve weightless networks learning in an unsupervised way. Using multivariate normal distributions (up to $n = 7$ dimensions) as training data, we obtained networks behaving much like a conventional neuron trained with Hebbian rule, i.e. displaying an orientation selectivity towards the direction of the greatest variance in data. Needless to say, some of these networks were learning faster and more accurately than conventional neurons with standard learning parameter settings (such as learning rate $\eta = 0.1$).

We also experimented with reinforcement learning in uncertain environments, evolving solutions to a simplified artificial bee problem [25] and some variants of T-maze problem [23, 26]. Also successful were experiments with evolving networks incorporating the principle of error backpropagation to learn in the well known intertwined spirals problem.

In principle, DSE allows to evolve any network topology with "aggregation nodes" (i.e. with indiscernible incoming connections) and n-way connections (explained below). To prove it, it is sufficient to consider node division, which can generate any number of uniquely labeled nodes (provided the set of labels is indefinite), and Con instruction, which can connect any two nodes; and which can be easily generalized to connect any number of nodes. Therefore the encoding is *complete* within the assumed space of topologies.

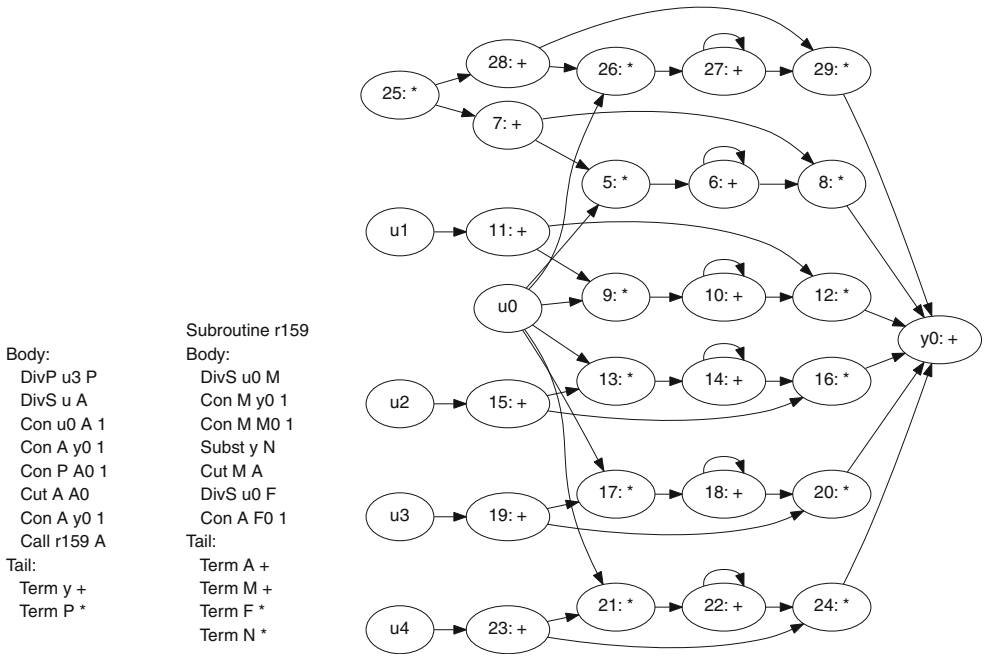


Fig. 7 Modular solution to the perceptron problem for $n = 4$ data inputs

The encoding is also *closed*, in that any genotype maps to a valid network; provided it is terminated properly during development.

DSE allows to employ arbitrarily defined node types, as well as connection types. They are encoded in the genotype and selected during development, but their functionality is left open. Given the object-oriented design of the network implementation, nodes and connections can be easily defined in terms of their evaluation and learning methods. Connections can also hold their own parameters, such as coefficients, or other structures, such as expression trees. Initialized connections, as constructed objects, appear in the arguments of Con* instructions and act as references during network development. So randomly initialized connections found in the genotype act as templates for connections in the growing network. It is easy to conceive, that such an approach allows to employ any learning rules—whether designed manually or generated randomly, e.g. by means of expression trees. Certainly the parameters in reference connections can be subjected to variation during reproduction of the genotype. This approach could also be applied to nodes, allowing their transfer functions to be evolvable.

A connection reference determines all the parameters of the connection, but obviously it can not specify the nodes to connect. These are specified in the remaining arguments of the Con instruction. A generalized Con instruction would require n arguments to specify n nodes to connect and 1 argument to determine the connection type, i.e. the reference. Although it is not clear, whether connections between 4 or 5 nodes would be ever considered, it is certainly interesting to consider connections between 3 nodes, i.e. with 1 additional (modulatory) input. The modulatory input can influence the dynamics of the connection, usually acting as a local reinforcement signal. Some research on the role of neuromodulation in artificial neural networks has been recently undertaken in [23, 25]. Next we show neuromodulation may also be employed within DSE.

4.4.1 Neuromodulation

Instructions Con and ConE create connections between two nodes. The internal dynamics of the connection (the evaluation as well as learning) can be defined arbitrarily, although with certain limitations. The evaluation and learning expressions must be local, i.e. involving only pre- and post-connection nodes and some local parameters only. This is sufficient to obtain some Hebbian rules of learning or even error backpropagation, but not some other potentially interesting rules. The concept of neuromodulation is to allow some third node to influence the dynamics of the connection, enabling the network to display much richer dynamics. Neuromodulation enables reinforcement Hebbian learning in the network, as for example in the following form [2]:

$$w(t+1) = w(t) + \eta r(t) x(t)[y(t) - w(t)], \tag{5}$$

where w is the connection weight, η is a learning rate, x and y are pre- and post-connection node values and r is the modulatory signal. In fact, neuromodulation allows to express the so called general correlative learning rule [2]:

$$w(t+1) = (1-\epsilon)w(t) + \eta x(t) r(t), \tag{6}$$

where ϵ is a forgetting rate, and r is a general learning value, which might be simply the post-connection node value, as in case of simple Hebbian learning; or some other reinforcement value, computed by the modulatory node.

Figure 8 shows a network solution for the perceptron problem (Sect. 4.3) using modulated connections. These are created using ConM instruction, which is the same as Con except it has a third argument to select the modulatory input node. The modulated connection is created whenever some nodes match the arguments of the instruction and no connection between the input and output nodes exists yet. Hence multiple connections are not allowed between nodes, but this is just a variant of implementation.

Regarding the phenotype, the error from the previous computational step is given by the u_0 input. Modulated connections, labeled with d, are defined to perform ordinary weighting during evaluation phase, and the following weight update during learning phase:

$$w(t+1) = w(t) + x(t-1) z(t), \tag{7}$$

where z is the modulatory input, i.e. u_0.

5 Experiments

DSE features scalability and modularity, hierarchy and ability to produce regular networks. It is difficult to assess, however, to what extent it possess these features. It is also difficult to quantify or even qualify differences between these features in DSE and in its two relatives—CE and HyperNEAT. An ultimate way to compare methods, is to compare their performance on specific problems. However, due to multiplicity of parameters and algorithmic details, many of which are undocumented, it is not a simple task to compare such complex methods.

Generality of these methods makes them applicable to a wide range of problems, so any benchmark on a few selected problems would be surely incomplete, and worse it would be inevitably flawed by the problem of parameter settings, i.e. the problem of how to set the parameters. Moreover, as these methods aim to solve problems of increasing complexity, it might not be easy to run

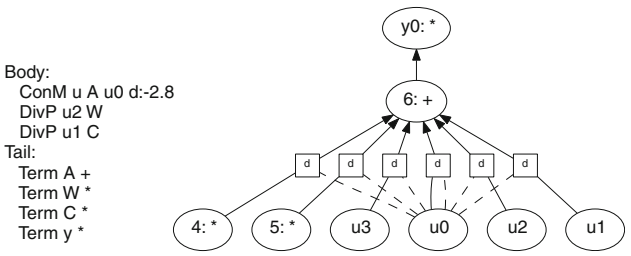


Fig. 8 Network solution for the perceptron problem ($n = 3$) using modulated connections

benchmarks even for these few problems due to computational demands.

Next, while complex methods are usually based on some fundamental concepts, they tend to be extended and developed into many variants. For example in CE there are plenty of so called encoding schemes, or instruction sets, to choose from; it can employ recurrence, automatically defined functions or learning during development. Likewise HyperNEAT is actively extended with new functionalities [5, 23, 24]; and DSE is not very different in this respect. Certainly there is nothing wrong with this, except it makes a conclusive comparison difficult, because any particular variant does not represent the whole method. This problem can be thought of as an extension of parameter settings problem. Finally, the comparison is further hampered by the lack of any commonly accepted "test suit", as in case of EAs for function optimization [see e.g. 32]. So the question of how to fairly and reliably compare these methods is wide open.

Nevertheless, in the remaining part of this section we conduct two modest experiments involving DSE, HyperNEAT and CE. Results indicate DSE is an interesting alternative to these encodings. Given the above considerations, however, the outcome can not be treated as anything more than just an indication.

5.1 Target connectivity problem

Both HyperNEAT and DSE aim to evolve regular and scalable networks. Regularity, however, is a matter of degree and complex problems, even those regular, might involve some irregularity as well. In [3], Clune et al. performed a set of experiments to examine capability of HyperNEAT to introduce exceptions into otherwise regular target topologies. The problem called Bit Mirroring was to establish some predefined connectivity pattern between two 2-dimensional layers of nodes. Each node in the input layer was preassigned some target node in the output layer and the fitness was proportional to the number of correctly wired nodes. The target connectivity pattern had a varying degree of regularity, with respect to columns and rows. From fully regular, where (i, j)-th node in the input layer had to connect to (i, j)-th node in the output layer, to quite irregular, where output nodes were selected randomly, either within column or within row or both. The experiment showed HyperNEAT was very successful in evolving highly regular connectivity patterns, however its performance deteriorated rapidly as regularity decreased. It was difficult for HyperNEAT to make exceptions in the regular patterns of connectivity.

We conduct a similar experiment here, in which we try to evolve target connectivity patterns between 1-dimensional layers of nodes. The problem is parametrized by three parameters: the size of layers n_u (i.e. the number of inputs/outputs), offset n_o, and the number of swaps n_s. The default connectivity is simply between all inputs u_i and outputs y_j satisfying $i = j$. Introducing the offset generalizes the relationship to:

$$j = (i + n_o) \bmod n_u, \;\; i = 0, \ldots, n_u - 1. \tag{8}$$

Finally, some target nodes are swapped. This is done by randomly selecting n_s pairs of input nodes (without repetition) and swapping their target nodes in the output layer. Much as in the original problem, fitness is proportional to the number of correctly wired output nodes; though actually we minimize the error.

We used HyperNEAT v3.0 C++ implementation[1] by J. Gauci, with all the parameters left default. The problem has been implemented identically in both systems. Worth mentioning is that evaluation does not depend on the phenotype network, since it can be calculated directly from weight (connectivity) matrix, which is binarized in case of HyperNEAT. We run the evolution for every combination of the following parameter values: $n_u = \{5, 10, 15\}$, $n_o = \{0, \ldots, 29\}$ and $n_s = \{0, 1, 2\}$—270 runs in total. Population size is 500 and the number of generations 300.

Figure 9 shows how the two encodings coped with the problem for the three different layer sizes. As can be seen, both systems have more troubles with larger problem instances, though DSE clearly outperforms HyperNEAT in absolute terms ($p < 0.001$, permutation test). It is more difficult to compare scalability for these data. One reasonable way would be to calculate how much longer it took to reach given level of fitness as the size of the problem increased. It took 5, 18 and 36 generations for DSE, to reach 0.6 fitness for $n_u = 5$, 10, 15, correspondingly. So it was $(36 - 18)/(18 - 5) = 1.38$ times longer on average to scale up from 10 to 15 than from 5 to 10. Analogous calculations for HyperNEAT yield $(196 - 88)/(88 - 23) = 1.66$. This indicates DSE also scales up better.

Another interesting question is how the two encodings managed the irregularity. Figure 10 (left) presents the most

[1] Available from http://eplex.cs.ucf.edu/hyperNEATpage/ at the time.

interesting case of $n_u = 15$. HyperNEAT performance gets visibly worse as the number of swaps increase. In contrast, irregularity have only a marginal impact on DSE. It might be argued this is a sign DSE does not generate the pattern in a regular manner, but builds the connectivity incrementally, as probably any direct encoding would do. But this is not true. Figure 10 (middle) presents an example of genotype generating solution network for $n_u = 15$, $n_o = 0$, $n_s = 2$ problem instance, where the swapped target nodes were: 14 with 9 and 11 with 10. The solution is almost perfect, as the evolution had plenty of "spare" generations to improve the solution in that run ($n_o = 0$ is the easiest case). In the first step of development, connection between u_9 and y_{14} is made, and then between u_{11} and y_{10}. In the 3rd and 4th steps, nodes y_{14} and y_{10} are relabeled with dummy terminal symbol @. Then a similar sequence is executed for y_9 and y_{11}. Finally the pattern of connections is established between inputs and outputs, which indexes satisfy $i = j$. Note that relabeled nodes are no longer in y layer, so they do not participate in the last operation. This way DSE found a regular connectivity pattern with four exceptions. Presented solution is not an exception, as many other solutions made effective use of ConE and CutE instructions, although also many produced the connectivity using Con instruction, i.e. on per-connection basis. Nevertheless, the experiment showed DSE is capable of setting regular patterns of connectivity and of handling exceptions far better than HyperNEAT. This is because it combines individual and patterned ways for establishing connections.

HyperNEAT's deficiency in producing regular networks with exceptions has been recently improved by extending it with a direct encoding called FT-NEAT. The resulting algorithm, HybrID, works by first finding a regular topology and then introducing exceptions. This hybrid approach improved HyperNEAT's results on three test problems [4].

We also run the experiment with our implementation of basic Cellular Encoding, using the following *encoding scheme*: {*ACYC*, END, PAR, SEQ, INCLR, DECLR, MRG, WAIT}, which guarantees completeness and closure within space of feed-forward topologies (see [9] for details). Using CE to solve the task turned out to be problematic, because it can not operate on inputs or outputs—the development tree can only operate on hidden nodes. Therefore we treated the hidden layer as an output layer, and in consequence the encoding not only had to find the target topology, but grow the layer as well (there was no penalty for excessive number of nodes though). Anyway, CE performed very poorly on this task. For $n_u = \{5, 10, 15\}$ it scored 0.93 ± 0.11, 1 ± 0, 1 ± 0 correspondingly, not producing any single solution, even in the easiest case. These results were significantly improved after throwing out SEQ, DECLR and MRG instructions from the scheme, but only in the $n_u = 5$ case—scoring 0.65 ± 0.26 and delivering two solutions.

Such an outcome might be predicted, as CE was never designed to produce regular patterns of connectivity, except of recurrent kind. An interesting exercise would be to construct a solution manually. We estimate it would require about n_u^2 properly placed instructions in the development tree and that figure is confirmed by the sizes of the two solution trees obtained, counting 31 and 32 nodes. At that rate of growth the genotype quickly becomes very brittle under genetic variation.

5.2 Retina problem

While HyperNEAT is evidently capable of producing regular topologies, it is unclear whether it can produce modular networks. Investigation into this issue has been recently done by Clune et al. [5]. Authors took the retina problem, originally proposed by Kashtan and Alon [15], as a suitable test for encoding's capability to evolve modular solutions.

The retina problem consists in evolving networks to recognize patterns on the left and right sides of an artificial retina, each side consisting of 2×2 pixels. Among 16 possible patterns on each side, half are considered positive, and they are symmetrical for the two sides. The task is to decide for all 256 possible pattern combinations whether

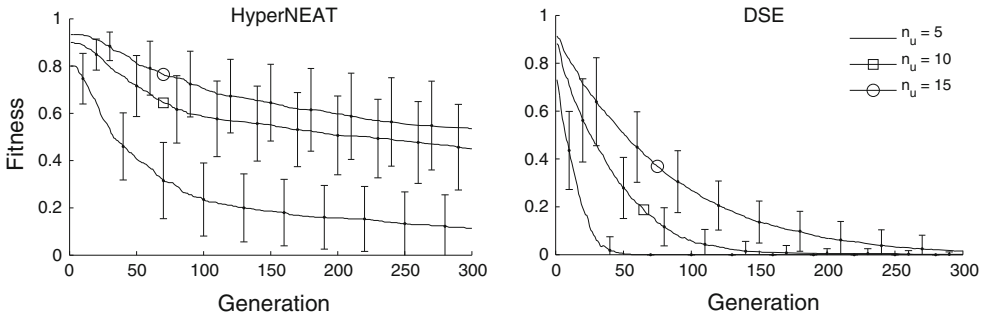


Fig. 9 Mean best fitness with standard deviation for both systems and different sizes n_u

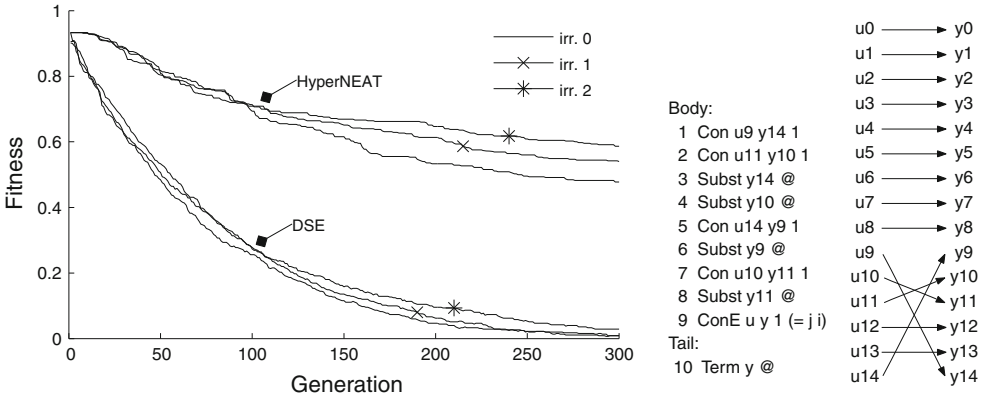


Fig. 10 Mean best fitness for $n_u = 15$ and increasing amount of irregularity for both systems (*left*); an example of solution genotype evolved by DSE for 0 offset, 2 swaps, 15 inputs task instance (*middle*) and the connectivity it produces (*right*)

either side contains a positive pattern (goal 'OR') or alternatively if both sides contain it (goal 'AND'). If the goal is fixed throughout evolution, nothing surprising happens—the evolution tries to solve the problem anyhow, usually producing networks intertwining both sides of retina. If the goal is periodically changing, however, it is no longer feasible to solve the problem efficiently without reflecting its structure. First, both sides of the retina shall be processed separately by the network solution, producing a response to positive patterns, which is a constant objective for both goals; and then results shall be combined appropriately, which is the changing part. The hypothesis behind the problem is that only encoding capable of generating modular networks can solve the changing variant of the problem efficiently. In fact, Kashtan and Alon [15] found, that modularly varying goal (MVG) not only enforced modular solutions, but actually allowed to solve the problem much faster. Here the fitness is a ratio of correctly classified patterns, usually falling between 0.75 and 1.0, and the network is considered a solution, if it scores 0.95.

The investigation conducted in [5] revealed a poor performance of HyperNEAT on the problem (see the paper for experimental setup details). In no scenario ('left only', OR, AND, MVG-20 and MVG-100, i.e. with the goal changing every 20 and 100 generations, respectively) median fitness of HyperNEAT's solutions exceeded 0.9 and no solutions have been reported on that problem, even after prolonging the evolution to 30,000 generations. The fitness function used in the experiment, however, was based on mean square error, which is not exactly a classification error.

We performed a similar experiment, using the same population size (500) and misclassification rate as a fitness function to be minimized. We used threshold transfer functions and weighted connections in case of DSE. The most important difference between DSE and HyperNEAT setups lied perhaps in the size and layout of the network, which was fixed in case of HyperNEAT and evolvable in case of DSE—the very fundamental difference for these systems. We also tested CE on this task, using threshold transfer functions and the same, 8-element encoding scheme as in previous experiment, except extended with instructions to manipulate connection weights and node biases. In all three systems networks were constrained to be feed-forward.

Figure 11 shows a median fitness of 20 evolutionary runs of DSE, lasting for 2,000 generations. Recognizing patterns only on the left side of retina was not difficult for DSE, as perfect solution was found in most runs. Scenarios AND, MVG-20 and MVG-100 all gave similar median around 0.1.

Table 3 compares the performance of DSE, HyperNEAT and CE in terms of mean best fitness (MBF, in %) with standard deviation from 20 evolutionary runs. The length of run was limited to 1,000 generations, mainly due to computational demands of HyperNEAT. In the table, we also show how many solutions were delivered by DSE, discerning three types of solutions: perfect ($f = 0$), standard ($f < 0.05$) and weak ($f < 0.1$). For example, it delivered 5 standard solutions for OR and 6 for AND goals in MVG-100 scenario. Figures for HyperNEAT are not displayed, because it delivered not a single, even weak solution. HyperNEAT was also significantly outperformed by DSE in terms of MBF ($p < 0.001$, permutation test). Even worse results in terms of MBF were produced by CE, although it succeeded in delivering 8 weak and three standard solutions in (and only in) the 'left only' scenario.

An important figure in Table 3 is the number of runs in which solutions were found for both goals in at least two consecutive periods ('cons.' entry) in MVG scenarios. Solving AND and OR goals in consecutive periods means the evolution is able to quickly switch between solution goals, which—according to Kashtan and Alon [15]—requires modular structure of the solution. DSE succeeded to do so in terms of standard solutions in two runs in case of MVG-20 and 3 in case of MVG-100, indicating DSE features that kind of modularity required to solve the retina

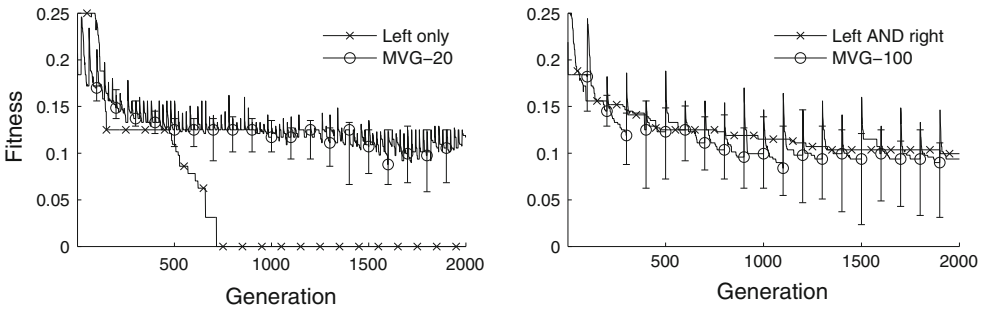


Fig. 11 Median fitness obtained with DSE for several scenarios of retina problem; in case of MVGs also 0.25 and 0.75 percentiles are shown

Table 3 Performance comparison of DSE, HyperNEAT and CE on retina problem

Scenario/goal	DSE				HyperNEAT	CE
	No. solutions with f					
	0	<0.05	<0.1	MBF [%]	MBF [%]	MBF [%]
Left only	13	13	13	4.4 ± 6.1	19.6 ± 2.6	12.5 ± 7.3
Left and right	0	1	7	10.5 ± 2.9	15.3 ± 0.8	23.1 ± 2.8
MVG-20 OR	0	2	6	10.2 ± 3.6	24.0 ± 0.9	24.3 ± 2.2
MVG-20 AND	0	3	8	9.8 ± 3.6	17.8 ± 0.9	21.3 ± 2.7
MVG-20 cons.	0	2	6			
MVG-100 OR	1	5	10	9.4 ± 4.9	23.9 ± 1.3	24.7 ± 1.2
MVG-100 AND	0	6	11	8.7 ± 4.1	17.3 ± 0.8	21.2 ± 3.1
MVG-100 cons.	0	3	10			

problem with varying goals. Analysis of these solutions confirms, that processing of both sides of retina is separated until the output node, where the final OR/AND processing step occurs. Unfortunately, obtained solutions are too large to be presented here. Their genotypes usually count more than 100 instructions and phenotypes more than 50 nodes.

6 Summary and conclusions

A novel developmental encoding for evolving networks has been proposed, called Developmental Symbolic Encoding. In this encoding, the genotype is a tree of routines, which in turn consist of lists of instructions saying how to develop the network. The network grows primarily by means of node divisions and connection arrangements, which is roughly how biological neural networks develop. DSE combines some concepts of CE and HyperNEAT. Much as CE, it can grow networks by means of node divisions, and features an explicit genetic modularity and hierarchy, conducing reuse of code and network modules. Much as HyperNEAT it can establish connectivity patterns between groups of nodes, and exploit some geometric-like relationships between them, enabling evolution of highly regular network topologies. These two ways of growing networks are combined in a coherent genetic representation, optimistically allowing to get best of both encodings while solving problems.

The encoding exhibits scalability—it can represent network phenotypes compactly, with the genotype growing slower than phenotype along the problem size. In other words, it is capable of capturing some regularities of network solutions, and thus regularities hidden in problems. The encoding has been also demonstrated to feature modularity and code reuse, where a single piece of genetic code, namely routine, generates multiple copies of subnetwork in the final network. Modularity and the evolution of modular solutions is supported by the fact, that routines are easily transferable between individuals and populations. This in turn opens an interesting further research on parallel multiple task solving, in which a number of populations solves a number of different, but related tasks, while possibly taking advantage of communication.

The scalability of DSE has been demonstrated in symmetry and parity problems. Evolved solutions for these problems were fully general with respect to the number of inputs, i.e. networks were able to automatically scale themselves up to the size of the problem during development, while being encoded by a fixed genotype. Certainly only for some problems such a perfect scalability can be achieved. Modularity, in turn, has been demonstrated in a more difficult variant of the parity problem and also in a classification problem. These problems required a capability to discover and exploit useful modules or subnetworks. DSE is also complete and closed, which means it

can represent any recurrent network topology and any genotype represents some valid network.

Also flexibility of the encoding has been shown, which allows to employ arbitrary node and connection types, including weightless, weighted, plastic and modulated connections. It is possible to employ nodes and connections having evolvable transfer and learning functions; restrict the space of topologies to feed-forward only; or impose any calculable constraints on the network, by including appropriate terms in a fitness function.

Much as CE and HyperNEAT, DSE is a complex method, involving many parameters and unspecified algorithmic details. There are endless options to modify the way things are done, or extend the encoding by new elements, such as instructions and genetic operators. From clarifying and simplifying the encoding, to introducing explicit learning algorithms, to extending the network model by non-aggregatory transfer functions. Further research over DSE is wide open.

Although it is difficult to reliably compare such (relatively) complex methods as HyperNEAT, CE and DSE, two experiments involving these encodings have been conducted. In the first one, DSE outperformed its relatives in evolving some predefined target connectivity patterns. Likewise in the second experiment, it gave the best results, delivering modular solutions to the retina problem. Thus we conclude DSE is a competitive neuroevolution method worth further development and trying in practice.

References

1. Balakrishnan K, Honavar V (2001) Evolutionary and neural synthesis of intelligent agents. In: Advances in the evolutionary synthesis of intelligent agents. The MIT Press, Cambridge, pp 1–27
2. Chen Z, Haykin S, Eggermont JJ, Becker S (2007) Correlative learning: a basis for brain and adaptive systems. Wiley-Interscience, New York
3. Clune J, Ofria C, Pennock R (2008) How a generative encoding fares as problem-regularity decreases. Parallel Problem Solving from Nature—PPSN X, pp 358–367
4. Clune J, Beckmann B, Pennock R, Ofria C (2009) HybrID: a hybridization of indirect and direct encodings for evolutionary computation. In: Proceedings of the European Conference on Artificial Life, Budapest, Hungary, pp 1–8
5. Clune J, Beckmann BE, McKinley PK, Ofria C (2010) Investigating whether HyperNEAT produces modular neural networks. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation, ACM, pp 635–642
6. Floreano D, Mattiussi C (2008) Bio-inspired artificial intelligence: theories, methods, and technologies. The MIT Press, Cambridge
7. Floreano D, Dürr P, Mattiussi C (2008) Neuroevolution: from architectures to learning. Evol Intel 1(1):47–62
8. Gauci J, Stanley KO (2007) Generating large-scale neural networks through discovering geometric regularities. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, New York, NY, USA, pp 997–1004
9. Gruau F (1994) Neural network synthesis using cellular encoding and the genetic algorithm. PhD thesis, Ecole Normale Suprieure de Lyon, France
10. Gruau F, Quatramaran K (1997) Cellular encoding for interactive evolutionary robotics. In: Fourth European conference on artificial life. The MIT Press, Cambridge, pp 368–377
11. Gruau F, Whitley D, Pyeatt L (1996) A comparison between cellular encoding and direct encoding for genetic neural networks. In: Proceedings of the first annual conference on genetic programming, MIT Press, Cambridge, pp 81–89
12. Harding S, Banzhaf W (2008) Artificial development. In: Organic computing, understanding complex systems, vol 21, Springer, Berlin, pp 201–219
13. Harding S, Miller J, Banzhaf W (2010) Developments in cartesian genetic programming: self-modifying CGP. Geneti Program Evol Mach 11(3):397–439
14. Hornby GS (2005) Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, pp 1729–1736
15. Kashtan N, Alon U (2005) Spontaneous evolution of modularity and network motifs. Proc Natl Acad Sci USA 102(39):13,773
16. Kitano H (1990) Designing neural networks using genetic algorithms with graph generation system. Complex Syst 4(4):461–476
17. Lipson H (2007) Principles of modularity, regularity, and hierarchy for scalable systems. J Biol Phys Chem 7(4):125
18. Mattiussi C, Floreano D (2007) Analog genetic encoding for the evolution of circuits and networks. IEEE Trans Evol Comput 11(5):596–607
19. McPhee N, Hopper N (1999) Analysis of genetic diversity through population history. In: Proceedings of the genetic and evolutionary computation conference, Citeseer, vol 2, pp 1112–1120
20. Montana D, VanWyk E, Brinn M, Montana J, Milligan S (2009) Evolution of internal dynamics for neural network nodes. Evol Intel 1(4):233–251
21. Newman MEJ, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69(2):26,113
22. Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Lulu Press, Raleigh
23. Risi S, Stanley KO (2010) Indirectly encoding neural plasticity as a pattern of local rules. In: From animals to animats 11, lecture notes in computer science, Springer, pp 533–543
24. Risi S, Lehman J, Stanley KO (2010) Evolving the placement and density of neurons in the HyperNEAT substrate. In: Proceedings of the 12th annual conference on genetic and evolutionary computation, ACM, pp 563–570
25. Soltoggio A, Dürr P, Mattiussi C, Floreano D (2007) Evolving neuromodulatory topologies for reinforcement learning-like problems. In: Proceedings of the IEEE congress on evolutionary computation, CEC, pp 2471–2478
26. Soltoggio A, Bullinaria J, Mattiussi C, Dürr P, Floreano D (2008) Evolutionary advantages of neuromodulated plasticity in dynamic, reward-based scenarios. In: Bullock S, Noble J, Watson R, Bedau MA (eds) Artificial life XI: proceedings of the eleventh international conference on the simulation and synthesis of living systems, MIT Press, Cambridge, MA, pp 569–576
27. Stanley KO (2007) Compositional pattern producing networks: a novel abstraction of development. Geneti Program Evol Mach 8(2):131–162

28. Stanley KO, Miikkulainen R (2002) Evolving neural networks through augmenting topologies. Evol Comput 10(2):99–127
29. Stanley KO, Miikkulainen R (2003) A taxonomy for artificial embryogeny. Artif Life 9(2):93–130
30. Stanley KO, D'Ambrosio DB, Gauci J (2009) A hypercube-based encoding for evolving large-scale neural networks. Artif Life 15(2):185–212
31. Verbancsics P, Stanley KO (2010) Transfer learning through indirect encoding. In: Proceedings of the 12th conference on genetic and evolutionary computation, ACM, pp 547–554
32. Whitley D, Rana S, Dzubera J, Mathias KE (1996) Evaluating evolutionary algorithms. Artif Intell 85(1–2):245–276
33. Yao X (1999) Evolving artificial neural networks. Proc IEEE 87(9):1423–1447