93–24 September 1993

RIJKSUNIVERSITEIT TE LEIDEN

VAKGROEP INFORMATICA

Designing Modular Artificial Neural Networks

Egbert J.W. Boers
Herman Kuiper
Bart L.M. Happel
Ida G. Sprinkhuizen-Kuyper

Department of Computer Science
Leiden University
Niels Bohrweg 1
P.O. Box 9512
2300 RA Leiden
The Netherlands

Designing Modular Artificial Neural Networks

Egbert J.W. Boers[i]
Herman Kuiper[ii]
Bart L.M. Happel[iii]
Ida G. Sprinkhuizen-Kuyper[iv]

Abstract

This paper presents a method for designing artificial neural network architectures. The method implies a *reverse engineering* of the processes resulting in the mammalian brain. The method extends the brain metaphor in neural network design with genetic algorithms and L-systems, modelling natural evolution and growth. It will be argued that a principle of *modularity*, which is inherent to the design method as well as the resulting network architectures, improves network performance.

1. Introduction

Several neural network simulation studies show that many problems can not be solved by a learning algorithm in conventional fully connected layered neural networks [e.g. 2, 5, 7, 10, 14, 18, 24]. Two problems that occur frequently are: a lack of generalization and a problem called interference. A trained network is said to generalize if it gives correct responses to input patterns not seen during training. Many networks fail to generalize because they have too many degrees of freedom. A large number of weights often allows a network to implement several different functions that correctly calculate the response to the patterns in the training set, but do not implement the desired function. One effect that can be seen in networks with this problem is *overtraining* (or overfitting [e.g. 8]). This effect happens when a small set of examples of the total task domain is trained for a very long time. The network initially learns to detect global features of the input, and as a consequence generalizes quite well. But after prolonged training the network will start to recognize each individual example input/output pair rather than settling for weights that describe the mapping for all cases in general. When that happens the network will give exact answers for the training set, but is no longer able to respond correctly for input not contained in the training set. This behaviour can be compared with curve-fitting with too many free parameters.

The problem of *interference* (or crosstalk) occurs if two or more unrelated problems are to be learned by one neural pathway [e.g. 11, 22]. With a very small network it may be that the network is simply not able to learn more than one of the problems. But if the network is large enough, in principle, to learn all tasks at the same time, it may still not be able to do so. The different problems seem to be in each others way: if one of the problems is represented in the weights of the network the other is forgotten, and vice versa. An example of such interference between more classifications is the recognition of both position and shape of an input pattern. Rueckl et al. [22] conducted a number of simulations in which they trained a three layer backpropagation network with 25 input nodes, 18 hidden nodes and 18 output nodes to simultaneously process form and place of the input pattern. They used nine, 3x3 binary input patterns at nine different positions on a 5x5 input grid. So there were 81 different combinations of shape and position. The network had to encode both *form* and *place* of a presented stimulus in the output layer. It appeared that the network learned faster and made less mistakes when the tasks were proc-

i. Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, email: boers@wi.LeidenUniv.nl, tel: +31-71-277093
ii. Research Engineer at the Knowledge Engineering Group of the National Aerospace Laboratory (NLR), Amsterdam, The Netherlands, email: kuiperh@nlr.nl.
iii. Department of Experimental and Theoretical Psychology, Leiden University, member of the Dutch Foundation for Neural Networks (SNN) and the Leiden Connectionist Group, email: happel@rulfsw.LeidenUniv.nl.
iv. Department of Computer Science, Leiden University, email: kuyper@wi.LeidenUniv.nl.

essed in separated parts of the network, while the total number of hidden nodes stayed the same. Of importance was the number of hidden nodes allocated to both sub-networks. When both networks had 9 hidden nodes the combined performance was even worse than that of the single network with 18 hidden nodes. Optimal performance was obtained when 4 hidden nodes were dedicated to the *place* network, and 14 to the apparently more complex task of the *shape* network. It needs to be emphasized that Rueckl et al. tried to explain why form and place are processed separately in the brain. The actual experiment they did, showed that processing the two tasks in one unsplit hidden layer caused interference. However, when experimenting with this example ourselves, we found that they failed to describe that removing the hidden layer completely, directly connecting the input and output layer, leads to an even better network than the optimum found using 18 hidden nodes in separate sub-networks.

In a follow-up paper [11] Jacobs et al. distinguished two types of crosstalk: spatial and temporal. Spatial crosstalk occurs when the output units of a network provide conflicting error information to a hidden unit. Temporal crosstalk occurs when units receive inconsistent training information at different times.

In order to eliminate these kind of problems a lot of research is being done on how to find the optimal network *architecture* given the problem it has to learn [e.g. 2, 5, 7, 8, 14, 24, 27]. A small example of how much depends on the architecture of a neural network is the XORproblem [e.g. 2, 23] in which a neural network has to learn to calculate the logical exclusive-or function. If the network has a simple layered architecture: two input units, two hidden units and one output unit, it takes on average 1650 epochs to train the network. If two additional connections are made, connecting the input directly to the output, only 30 epochs are needed to achieve the same residual error[i]. This paper will present a network design method that is largely inspired by the way in which the human brain is believed to have evolved.

1.1 Learning as entropy reduction

A neural network, when trained, is performing an *input-output mapping*. The mapping that is implemented by the network depends on the architecture of the network and its weights. When the architecture is fixed, the mapping of the network is solely determined by the weights. The set of all possible weight configurations (*weight space*) of a network determines a *probability distribution* over the space of possible input-output mappings that can be implemented with the fixed architecture. The *entropy* of this distribution is a quantitative measure of the diversity of the mappings realizable by the architecture under consideration (see e.g. [4, 8, 24]).

Learning from examples reduces the intrinsic entropy of the untrained network by excluding weight configurations which realize mappings incompatible with the training set. The residual entropy of the trained network is a measure of its generalization. The goal of this research has been to design a method to find a network architecture for a given task, that has a residual entropy approaching zero after training from examples, which means that the network converges to a state that implements the desired mapping independent of the initial weight configuration before training. If a network architecture is able to give several different mappings as a result of the training from examples, the residual entropy will not be zero. This will probably lead to a bad generalization because only *examples* of the complete domain are used for training, and it is possible to extract the same set of examples from many different mappings. An optimal architecture of a network will only allow for the mapping that is actually wanted. If the residual entropy after training is not zero, small deviations from the input patterns in the training set can give large errors in the output. This can happen, for example, when we have a weight space with a large number of minima, each implementing exactly the mapping of the training set, but just one (or none!) representing the mapping actually wanted.

Important is also the selection of the training set, because the error surface of the weight space depends on the training data. Different training sets can result in quite different error surfaces. Minima of the

i. We used backpropagation with a momentum term. Our experiments showed that the actual setting of the parameters did not matter very much if the architecture was optimal. The 'standard' 2-2-1 network performed optimal with a learning rate parameter of 0.6 and a momentum term of 0.65.

error surface at the same location in weight space, that appear with different training sets can be expected to show a better generalization.

1.2 Definition of entropy

In order to give a definition of the entropy of a neural network, the probability of a network implementing a mapping f has to be defined. Given a network architecture and its corresponding weight space W (the set of all possible weight settings w), Solla [24] defines the *a priory probability* of the network for a mapping f as:

$$P_f^0 = \Omega_f / \Omega_W,$$

where Ω_W equals the total volume of the allowed weight space, and

$$\Omega_f = \int_{\Omega_W} \Theta_f(w)\, dw$$

is the volume in the initial weight space that implements the desired mapping with

$$\Theta_f(w) = \begin{cases} 1 & \text{if } w \text{implementsmapping} \quad f \\ 0 & \text{otherwise} \end{cases}$$

So in other words: P_f^0 is the probability the network implements the desired mapping with an arbitrary setting of w. Obviously the network should be able to implement the desired mapping: $P_f^0 \neq 0$. If $P_f^0 = 0$, the network is unable to learn the desired input/output mapping.

Selecting a network structure defines a class of functions that are realizable with that network. A useful measure of the diversity of possible mappings f that can be implemented with the chosen architecture is the a priory *entropy* [4]:

$$S^0 = -\sum_{\{f\}} P_f^0 \ln(P_f^0)$$

of the a priory probability distribution of a given network. Since $P_f^0 \neq 0$ not just for the desired mapping f but also for mappings $f' \neq f$, the distribution of P_f^0 is such that the entropy $S^0 > 0$.

The definition of the *a priory probability* and the *entropy* of neural networks give a mathematical notion of the necessity to find networks with a good architecture. A correct network architecture results in a high a priory probability, which in turn leads to a low intrinsic entropy of the untrained network. It is the intrinsic entropy S^0 of the untrained network that needs to be eliminated via a learning process. The purpose of training is to confine the configuration space to the region $\{w \| \Theta_f(w) = 1\}$, thus eliminating all ambiguity about the input-output mapping implemented by the trained output. Note that a high a priory probability of implementing the wanted function does not imply that the learning algorithm is always able to find it.

Network entropy, however, can be reduced by fairly straightforward architectural constraints, improving the network's capability to learn a desired I/O mapping [5, 24]. It is a major difficulty that thus far there are no general methods or guidelines providing useful architectural constraints.

1.3 Reverse engineering

The human brain itself appears to have a modular organization [1, 17, 25] closely related to its neural functions [7, 12, 18, 19]. This modular organization appears already to be partially present at birth, and is assumed to *direct* learning [7]. The modules on the smallest scale, of which most of the cortex is built, can be seen as the elementary computing units of the brain. These modules, often called *minicol-*

umns [17, 25], consist of approximately 100 neurons. It is hypothesised [13] that the eventual functionality of each module is determined by the afferent signals received during growth and learning.

The actual architecture of each module and the overall structure of the brain has to be genetically coded in some way and is expressed in a growth process during the development of the brain. The evolutionary process must have discovered that it was profitable just to make more and more of the same cortical units. That this is what actually happened can be concluded from the fact that the architecture of the basic unit does not vary very much across species and cortical areas [13]. This kind of repeating the same module over and over can be seen almost everywhere in nature [e.g. 3].

We propose a reverse engineering of the natural processes that resulted in the brain, to design artificial neural network architectures that capture the functional and structural characteristics of biological neural systems. To this aid, a model used to describe growth in nature: L-systems [15, 16, 20, 21], is combined with genetic algorithms [6, 9] in an attempt to extend the brain metaphor in neural network design with genetic search and artificial growth. Results of experiments with this method performed so far indicate this method to be very promising [2].

2. L-systems

L-systems are a mathematical construct developed to model the biological growth of plants [15] and can be seen as a special class of fractals [16]. L-systems are based on a parallel string rewriting mechanism. The resulting strings, after repeatedly applying production rules on an initial axiom, can be interpreted in several ways depending on the semantics of the symbols used [15, 20, 21]. The most used interpretation is used to generate line drawings in the shape of trees. To describe graph topologies[i], an interpretation was constructed that enabled context-sensitive graph rewriting. The use of L-systems for generating neural network architectures is expected to introduce self-similar and modular architectural constraints, repeatedly and recursively reusing useful neural processing principles.

2.1 Strings coding topology

The strings used in this research consist of characters from the rather arbitrarily chosen alphabet {A–H, 0–5, [,]}. A node from the network is represented by a letter from this alphabet, and can be seen as the smallest possible module. Larger modules can be created by grouping nodes (or other modules) between square brackets. Connections are made by inserting digits, or *skips*, between the modules. When a digit x is encountered, the preceding module is connected to the module that is x skips to the right. Since we used the backpropagation algorithm to test the quality of the network architecture represented by the string, only *feedforward* networks were generated. Recurrent network architectures can easily be generated by allowing negative skips. All *output* nodes of the first of the two connected modules are then connected to all *input* nodes of the second. An input node of a module is a node that receives no input from within the module. An output node of a module has no output to other nodes within the module. If a skip from within a module goes beyond the closing bracket of this module, the skip is continued after this bracket. As treated below, this specific way of coding network architectures allows for a repeated and recursive application of good production rules. It is expected that architecture constraints beneficial on a small scale, will also benefit performance when applied to large-size structures. This reusing of the same rules results in fractal-like network architectures.

In the alphabet we used, the maximum skip distance is 5, which restricts the kind of networks that are possible. It is easy to prove that allowing infinite skip distances makes it possible to describe all possible graphs with the strings described above. It is unknown whether this limitation seriously restricts the quality of the possible solutions, and for small problems it restricts the search space of the genetic algorithm hereby speeding up convergence.

i. In this work graph *topologies* are used to describe network *architectures*.

1:			A			$\rightarrow$ B0B0B
2:			B	>	B	$\rightarrow$ [CD]
3:			B			$\rightarrow$ C
4:	C	<	D			$\rightarrow$ C
5:			D	>	D	$\rightarrow$ C1

Fig. 1 *Production rules*

2.2 Production rules

The strings described in the previous section are generated by applying production rules to an initial string (axiom). The L-systems we used are 2L-systems: every production rule can have both a left and a right context. The production rules have the following format:

$$L < P > R \rightarrow S.$$

The four elements of the production rule can not be arbitrary strings from our alphabet; we made the following restrictions:

- The *predecessor* P may contain skips and modules, and should contain at least one letter.
- These same restrictions apply to the *successor* S, which is also allowed to be empty.
- The *contexts* L and R may also be empty, but no loose digits are allowed: each digit must be contained in a module.

A deviation from conventional 2L-systems is the handling of context, that now has to take the skips into account. Whereas in conventional 2L-systems the context is matched at the characters directly to the left and right of the predecessor, on our L-system the context is determined by looking at the actual network that is coded by the string that is being rewritten. Because of this, the context string should be seen as a list of modules, all of which should be connected to the predecessor in the current network before the production rule may be applied. The left context of the predecessor is the set of modules or nodes that give input to (parts of) the predecessor, the right context is defined in an analogous way. Furthermore, we used the convention that production rules with larger matching contexts prevail.

For example, look at the production rules shown in figure 1. If A is taken as axiom, the rewriting process is as follows:

A $\rightarrow$ B0B0B $\rightarrow$ [CD]0[CD]0C $\rightarrow$ [CC1]0[CC]0C.

Hereafter no more rules apply. The successive networks of this growth process are shown in figure 2. It is very easy to make production rules that can be applied infinitely, so that the growing process never stops. For example the rule A $\longrightarrow$ AA will grow exponentially. For this reason we restricted the maximum number of steps to 6.

The strings that resulted from the L-system in our simulations were transformed to adjacency-matrices at every step in the rewriting process in order to be able to calculate the left and right contexts for the next step. The adjacency-matrix that resulted from the last rewriting step was pruned to remove unconnected and useless nodes.

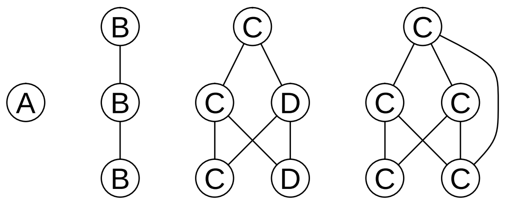


Fig. 2 *The networks*

3. Genetic Algorithms

The Neo-Darwinian theory of evolution is modelled with genetic algorithms [e.g. 3, 6]. A population of strings is manipulated, where each string can be seen as a chromosome (the genotype), consisting of a number of genes. These genes are used to code the parameters for a problem. Each string can be assigned a fitness, which indicates the quality of the solution (the phenotype) it encodes. The strings used by the algorithm reproduce proportional to their fitness. A new generation is created by selecting and recombining existing strings based on their fitness, using genetic operators like selection, crossover, inversion and mutation. From generation to generation the mean fitness of the population should increase. Genetic algorithms have proved to provide a powerful method for solving multiple constraint problems. The specific genetic algorithm used is derived from the GENITOR algorithm [26, 27] that uses a static population model. In this model, that uses rank based selection, each new member of the population is inserted in the population according to its fitness, removing the population's worst member and keeping the population sorted according to fitness.

3.1 Coding the production rules

Searching efficient network architectures proceeds by having a genetic algorithm define and optimize the production rules of an L-system. Here, a chromosome encodes a set of production rules (the L-system). Each character from the alphabet was represented with a six bits long binary string. The 17 symbols that were coded include the 16 characters from our alphabet plus a special symbol (an asterisk), used to separate the constituent parts of production rules (context, predecessor, successor) within a chromosome. The asterisk may be compared to the start and stop markers used for the transformation from RNA to protein. Production rules are extracted from a chromosome according to the following procedure:

1. Read the chromosome, six bits at a time.
2. Translate each group of six bits to an asterisk or a symbol of our alphabet, according to a translation table. This results in a string of symbols.
3. Find all minimal substrings containing 5 asterisks of this symbol string. These substrings begin and end with a asterisk, and have three asterisks somewhere in between.
4. Each substring now codes one production rule. The symbols of our alphabet between two asterisks form a part of the production rule. For example **A*BBB*C* codes the rule A > BBB $\rightarrow$ C. Notice that two asterisks next to each other indicate an empty part of the production rule.
5. Throw away all production rules that do not conform to the restrictions given in paragraph 2.2, leaving only valid production rules.
6. Repeat steps 1–5 by starting to read the bitstring not just at the first bit, but also at bit 2–5.
7. Repeat steps 1–6, starting at the end of the bitstring, reading the bits in the opposite direction.

All production rules that are extracted in this way form the L-system for one network. Since our algorithm starts at *all* bit positions and reads in *both* directions, the chromosome of one member of the population is read *twelve times*, which may eventually increase the level of implicit parallelism of the genetic algorithm. It even allows two or more production rules to be coded by the same bits.

3.2 Evolution

The genetic algorithm generates a population of initially random bit strings. Each bitstring (chromosome) is a member of the population. A solution is evaluated by having the decoded L-system produce the network architecture. A neural network simulator (in this research the backpropagation algorithm) then trains the network for a specified problem. The obtained residual error after a certain training period is then transformed into a measure of fitness. This fitness is returned to the genetic algorithm, which produces a new solution from the population to be evaluated, etcetera.

One should however be very careful with the interpretation of the solution to which the genetic algorithm converges. Since the parameters of the learning algorithm are not also subject to the optimization process, the obtained best network architecture depends on these parameters. If, for example, in order

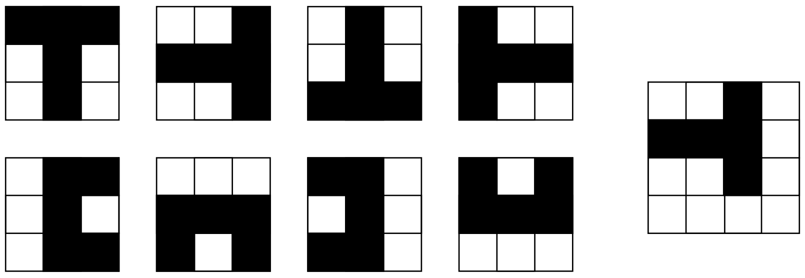


Fig. 3 *The 8 possible letters and one sample input grid of 4x4.*

to reduce the execution time of the evolutionary process the number of training epochs is reduced to 100, say, it may very well be that the resulting network architecture performs much worse than an other network architecture when the training time is extended. A possible solution might be to include these parameters in the evolutionary process, but that will increase the search space for the genetic algorithm considerably.

4. Results

One of the problems that were presented to the outlined method is the TC-problem [23]. The network has to learn whether a T or a C is presented to it. Each letter, consisting of 3x3 pixels, can be rotated 0, 90, 180, or 270° and can be anywhere on the 4x4 input grid, see figure 3. The network that resulted after 7500 genetic recombinations is shown in figure 4. Notice that the network has just 13 instead of 16 input nodes, the last three positions in the grid were not used. We compared this network architecture with networks with one hidden layer, varying the number of hidden nodes from 3–8[i]. We presented the 32 patterns 250 times to the networks, and repeated this 50 times after resetting the networks with random initial weights[ii]. Of these 50 times, the evolved network of figure 4 did classify all 32 patterns correctly in 45 cases, where the network with one hidden layer with 6 nodes did classify correctly in only 30 cases. The other networks we tried were even worse.

A more difficult task for backpropagation is derived from [10]. In this research it was tried to have a neural network correctly classify a 10x10 map into four classes using backpropagation, see figure 5. The input of this problem consisted of two values between 0 and 1, defining a coordinate in a plane. The network had to classify the input values into four classes. This only succeeded for the map he used

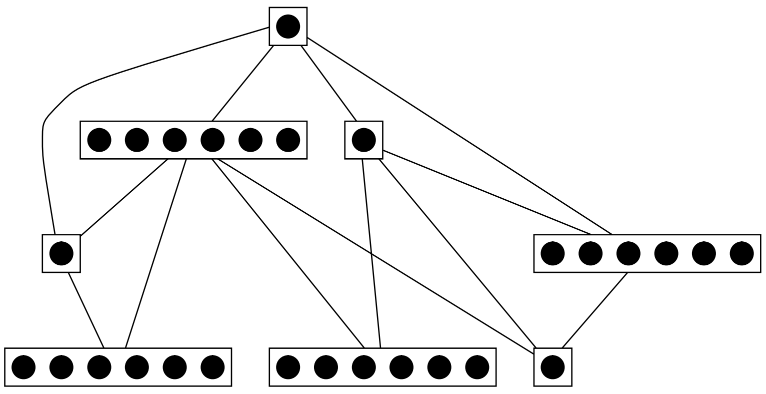


Fig. 4 *Network for the TC-problem*

i. We used backpropagation with momentum. The learning parameter was set to 0.4 and the momentum parameter was set to 0.9.

ii. The incoming weights of each module were initialized with values chosen from a uniform distribution from the interval $[-3/\sqrt{n}, 3/\sqrt{n}]$, where n is the *fan-in* of the module, which prevents the learning algorithm to start from one of the many plateaus in weight-space.

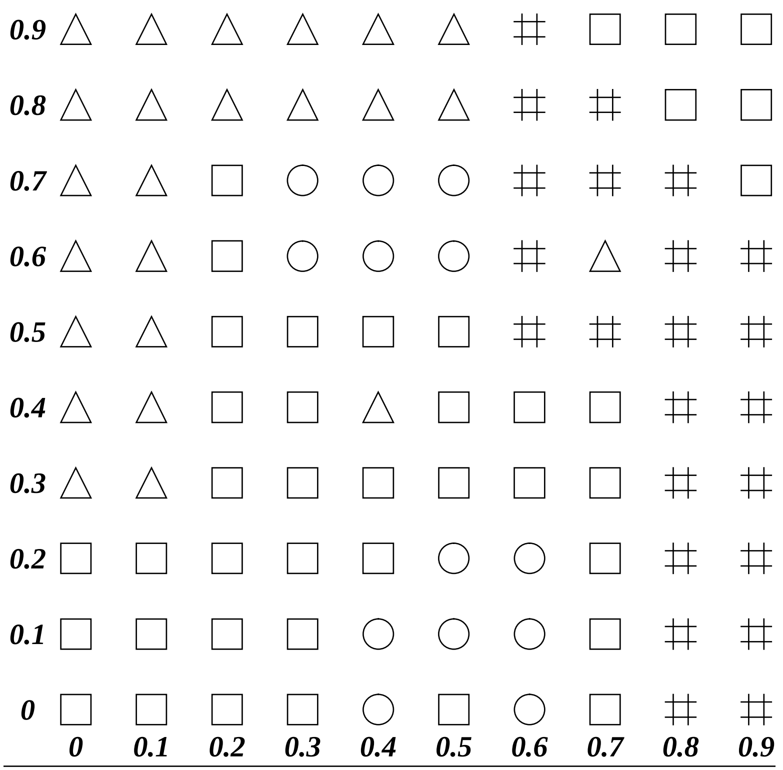
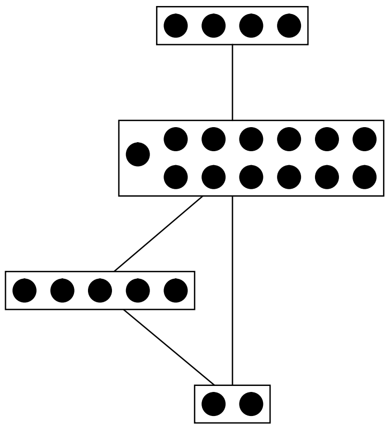


Fig. 5 *The input map*

with a network of 3 hidden layers with 20 nodes each. This network frequently reached a local optimum. Using our design method we found a much smaller network architecture that learned the problem significantly faster (the network converged in 100 epochs on average). The resulting network is shown in figure 6. It could correctly classify 99 of the 100 points, each time the problem was presented to it.

5. Conclusion

Our simulation results indicate that the combined use of genetic algorithms and L-systems results in an efficient search, characterized by fast convergence towards a solution and better architectures. This might be explained by a number of theoretical advantages of the use of L-systems to code network



A→[BCBA]5A4
B→AC
A>CC→

Fig. 6 *Network and rules*

topologies over ‘blueprint representations’ where the genetic algorithm has to specify every single connection:

1. Coding can be sparse (less free parameters). A few production rules can produce already very complex architectures.
2. Scalability of solutions. Arbitrarily large architectures can easily evolve from small architectures by relatively minor changes in a fixed number of production rules. This significantly reduces the time needed for genetically optimizing large architectures, see also [14].
3. Modularity of solutions. The same production rules can be applied many times in the growth process resulting in the multiple application of efficient pieces of architecture, and self-similar fractal-like architectures.
4. This method of encoding network topologies is much related to the way biological architectures are genetically encoded and might therefore be expected to provide some efficient design principles.

References

[1] D.A. Allport; ‘Pattern and actions’. In: *New directions in cognitive psychology,* G.L. Clagton (Ed.), Routledge and Kegan Paul, London, 1980.

[2] E.J.W. Boers and H. Kuiper; *Biological metaphors and the design of modular artificial neural networks*. Unpublished Master’s thesis, Leiden University, 1992.

[3] R. Dawkins; *The blind watchmaker*, Longman, 1986. Reprinted with appendix by Penguin, London, 1991.

[4] J.S. Denker, D.B. Schwartz, B.S. Wittner, S.A. Solla, R.E. Howard, L.D. Jackel and J.J. Hopfield; ‘Large automatic learning, rule extraction and generalization’. In: *Complex systems*, 1, 877-922, 1987.

[5] N. Dodd; ‘Optimization of network structure using genetic algorithms’. In: *Proceedings of the International Neural Network Conference,* INNC-90-Paris, 693–696, B. Widrow and B. Angeniol (Eds.), Kluwer, Dordrecht, 1990.

[6] D.E. Goldberg; *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, Reading, 1989.

[7] B.L.M. Happel and J.M.J. Murre; ‘The design and evolution of modular neural network architectures’. (in prep.)

[8] J. Hertz, A. Krogh and R.G. Palmer; *Introduction to the theory of neural computation*. Addison-Wesley, Redwood City, 1991.

[9] J.H. Holland; *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Harbor, 1975.

[10] R.J.W. van Hoogstraten; *A neural network for genetic facies recognition*. Unpublished student report, Leiden, 1991.

[11] R.A. Jacobs, M.I. Jordan and A.G. Barto; ‘Task decomposition through competition in a modular connectionist Architecture: the what and where vision tasks’. In: *Cognitive Science*, 15, 219–250, 1991.

[12] E.R. Kandel and J.H. Schwartz; *Principles of neuroscience*. Elsevier, New York, 1985.

[13] H.P. Killackey; ‘Neocortical expansion: an attempt toward relating phylogeny and ontogeny’. In: *Journal of cognitive neuroscience*. 2, 1–17, 1989.

[14] H. Kitano; ‘Designing neural networks using genetic algorithms with graph generation system’. In: *Complex Systems*, 4, 461–476, Champaign, IL, 1990.

[15] A. Lindenmayer; ‘Mathematical models for cellular interaction in development, parts I and II’. In: *Journal of theoretical biology*, 18, 280–315, 1968.

[16] B.B. Mandelbrot; *The fractal geometry of nature*. Freeman, San Francisco, 1982.

[17] V.B. Mountcastle; ‘An organizing principle for cerebral function: the unit module and the distributed system’. In: *The mindful brain*, G.M. Edelman, V.B. Mountcastle (Eds.), MIT Press, Cambridge, MA, 1978.

[18] J.M.J. Murre; *Categorization and learning in neural networks. Modelling and implementation in a modular framework*. Dissertation, Leiden University, 1992.

[19] M.I. Posner, S.E. Peterson, P.T. Fox and M.E. Raichle; ‘Localization of cognitive operations in the human brain’. In: *Science*, 240, 1627–1631, 1988.

[20] P. Prusinkiewicz and J. Hanan; *Lindenmayer systems, fractals and plants*. Springer-Verlag, New York, 1989.

[21] P. Prusinkiewicz and A. Lindenmayer; *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990.

[22] J.G. Rueckl, K.R. Cave and S.M. Kosslyn; ‘Why are “what” and “where” processed by separate cortical visual systems? A computational investigation’. In: *Journal of cognitive neuroscience*, 1, 171-186, 1989.

[23] D.E. Rumelhart and J.L. McClelland (Eds.); *Parallel distributed processing. Volume 1: Foundations*. MIT Press, Cambridge, MA, 1986.

[24] S.A. Solla; ‘Learning and generalization in layered neural networks: the contiguity problem’. In: *Neural networks: from models to applications,* 168–177, L. Personnas and G. Dreyfus (Eds.), I.D.S.E.T, Paris,

1989.

[25] J. Szentagothai; ‘The “module-concept” in the cerebral cortex architecture’. In: *Brain Research*, 95, 475–496, 1977.

[26] D. Whitley; ‘The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best’. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their applications (ICGA)*, 116–121, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.

[27] D. Whitley and T. Hanson; ‘Towards the genetic synthesis of neural networks’. In: *Proceedings of the 3rd International Conference on Genetic Algorithms and their applications (ICGA)*, 391–396, J.D. Schaffer (Ed.), Morgan Kaufmann, San Mateo CA, 1989.