Chapter 8
Neuro-Centric and Holocentric Approaches to the Evolution of Developmental Neural Networks

Julian F. Miller

Abstract In nature, brains are built through a process of biological development in which many aspects of the network of neurons and connections change and are shaped by external information received through sensory organs. From numerous studies in neuroscience, it has been demonstrated that developmental aspects of the brain are intimately involved in learning. Despite this, most artificial neural network (ANN) models do not include developmental mechanisms and regard learning as the adjustment of connection weights. Incorporating development into ANNs raises fundamental questions. What level of biological plausibility should be employed? In this chapter, we discuss two artificial developmental neural network models with differing degrees of biological plausibility. One takes the view that the neuron is fundamental (neuro-centric) so that all evolved programs are based at the level of the neuron, the other carries out development at an entire network level and evolves rules that change the network (holocentric). In the process, we hope to reveal some important issues and questions that are relevant to researchers wishing to create other such models.

1 Introduction

Although artificial neural networks (ANNs) are over 60 years old [1] few would argue that they even approach the learning capabilities of relatively simple organisms. Yet, over this period our understanding of neuroscience has increased enormously [2] and computer systems have gained enormous improvements in speed. We suggest that a major weakness of many ANNs models is that they follow the "synaptic dogma" (SD) which encodes learned knowledge solely in the form of connection strengths

J. F. Miller (✉)
Department of Electronics, University of York, York, UK
e-mail: julian.miller@york.ac.uk

T. Kowaliw et al. (eds.), *Growing Adaptive Machines*,
Studies in Computational Intelligence 557, DOI: 10.1007/978-3-642-55337-0_8,

(i.e. weights). SD gives rise to "catastrophic forgetting" (CF) [3–5]. This is where trained ANNs when re-trained on a new problem, forget how to solve the original problem. Interestingly, although Judd showed that learning the weights in a fixed neural network is an NP-complete problem [6], Baum proved that if one allows the addition of neurons and weighted connections, ANNs can solve learning problems in polynomial time [7]. Shortly after Baum's paper "constructive ANNs" were devised [8, 9]. These use supervised learning, in which neurons and connections are added gradually and weights adjusted incrementally until training errors are reduced. Quinlan discussed "dynamic networks" which he defined as "any artificial neural network that automatically changes its structure through exposure to input stimuli" [10, 11]. He observed that "although there has been some work on the removal of weighted connections, these schemes have not explored a general framework where the number of connections is both increased and decreased independently of the number of processing units" [10]. Combining evolution with development is a way of exploring the more dynamic networks that Quinlan discussed. However, despite the fact that the original inspiration for ANNs came from knowledge about the brain, it still remains that very few ANN models use both evolution and development, both of which are fundamental to the construction of the brain [12].

Apart from the problem of catastrophic forgetting, there is much research that undermines the notion that memory in brains is principally related to synaptic strengths. Firstly, it is now known that most synapses are not static but are constantly pruned away and replaced by new synapses and learning is related strongly to this process [13]. Secondly, synaptic plasticity does not merely involve the increase or decrease of the number of synapses, but is related also to the exact location of the synapses on the dendritic tree and the actual geometry of the dendritic branches. Thirdly, much research indicates that learning and environmental interaction are strongly related to *structural* changes in neurons. Dark-reared mice when placed in the light develop new dendrites in the visual cortex within days [14]. Animals reared in complex environments involving active learning have an increased density of dendrites and synapses [15, 16].[1] Within the brains of songbirds in the breeding season, it is has been found that the number, size and spacing of neurons increases [18]. In a well-known study it was found that the hippocampi of London taxi drivers who must remember large parts of central London, are significantly larger relative to those of control subjects [19]. Rose is quite emphatic about the role of structural change in memory and argues that after a few hours of learning the brain is permanently altered "if only by shifting the number and position of a few dendritic spines on a few neurons in particular brain regions" [20]. Another, almost obvious, aspect supporting the view that structural changes in the brain are strongly associated with learning, is simply that the most significant period of learning in animals happens in infancy, when the brain is developing [21].

[1] However, a recent study showed that environmental enrichment alone does not significantly increase hippocampal neurogenesis or bestow spatial learning benefits in mice [17].

Our view is that structural changes in a computational network will enhance the learning capability of artificial neural networks and development appears to be a promising way of acheiving this. The idea is that evolution will arrive at useful developmental rules that result in enhanced learning ability. In this chapter we discuss two models that attempt to do this. The first is neuro-centric and uses evolution to build a collection of programs that represent many aspects of a neuron (i.e. dendritic and axonal branches, soma, synapses and neuron firing) [22–28]. The advantage this has is that many aspects of neuroscience can enrich and inform such an approach. The second approach is holocentric, it uses evolution and development, however evolution works at the whole system level. Most conventional ANNs are also holocentric as learning mechanisms are employed at a whole network level. Holocentric models tend to use a much higher level of abstraction of neural systems and consequently they are simpler and computationally more efficient. The holocentric model we discuss here uses a genotype which unfolds through self-modification and iteration to produce an entire computational network. At each iteration a complete functioning network is obtained and embedded self-modification operators dictate changes to the network so that a new network is produced. In both models the computational network develops at run time, so that if executed for sufficiently long periods of time, the networks can become arbitrarily large and complex. This contrasts with other well known developmental computational approaches such as HyperNEAT [29] and Cellular Encoding [30] where the phenotype is fixed and defined by artificial evolution.

The neuro-centric method uses a form of genetic programming (GP) [31] called Cartesian Genetic Programming (CGP) [32, 33] to represent the computational components of the neuron. It is referred to as the CGP developmental network (CGPDN). The other approach (holocentric) is based on a form of CGP called Self-Modifying CGP (SMCGP)[34–40]. CGP is a method for evolving graph-based computational structures so it naturally lends itself to representing ANNs. Indeed, ANNs can be encoded directly by CGP and recent work suggests this is in itself a very promising encoding [41–43]. CGP has been shown to offer advantages over other forms of GP, in that it can evolve solutions in less genotype evaluations and solutions are more compact than many other methods [44]. Like GP it is a general method for evolving programs so that it provides a method for evolving many kinds of computational structures (e.g. Boolean circuits, computer programs, sets of equations) including artificial neural networks.

The plan of the chapter is as follows. Since CGP underlies the models discussed here, we give an overview in Sect. 2. In Sect. 3 we outline a neuro-centric CGP developmental network (CGPDN). In Sect. 4 we discuss a developmental form of CGP called self-modifying CGP. In the process we compare and contrast the developmental CGP approach with the non-developmental approach. In Sect. 7 we discuss a developmental holocentric approach to ANNs that uses SMCGP to develop ANNs. We discuss and contrast the neuro-centric and holocentric approaches to ANNs in Sect. 8. We end the chapter with some observations on requirements for acheiving general learning in ANNs.

2 Cartesian Genetic Programming

For completeness we give a brief overview of CGP. A more detailed account is available in the recently published book [45]. In CGP, programs are represented in the form of directed acyclic graphs. These graphs are represented as a two-dimensional grid of computational nodes (which may, or may not, be sigmoidal neurons). The genes that make up the genotype in CGP are integers that represent where a node gets its data, what operations the node performs on the data and where the output data required by the user is to be obtained. When the genotype is decoded, some nodes may be ignored. This happens when node outputs are not used in the calculation of output data. When this happens, we refer to the nodes and their genes as 'non-coding'. We call the program that results from the decoding of a genotype a phenotype. The genotype in CGP has a fixed length. However, the size of the phenotype (in terms of the number of computational nodes) can be anything from zero nodes to the number of nodes defined in the genotype. The types of computational node functions used in CGP are decided by the user and are listed in a function look-up table. For instance, if conventional ANNs are required there might be only one function, a sigmoid. In this case function genes are not required as by default they are all sigmoid. However, in general CGP can use any combination of functions desired. In [41–43] two functions are used, sigmoid and hyperbolic tangent.

In CGP, each node in the directed graph represents a particular function and is encoded by a number of genes. One gene is the address of the computational node function in the function look-up table. We call this a *function gene*. The remaining node genes say where the node gets its data from. These genes represent addresses in a data structure (typically an array). We call these *connection genes*. Nodes take their inputs in a feed-forward manner from either the output of nodes in a previous column or from a program input. It should be noted that recurrent networks can also be represented as in [41].

The number of connection genes a node has is chosen to be the maximum number of inputs (often called the arity) that any function in the function look-up table has. The program data inputs are given the absolute data addresses 0 to $n_i - 1$ where n_i is the number of program inputs. The data outputs of nodes in the genotype are given addresses sequentially, column by column, starting from n_i to $n_i + L_n - 1$, where L_n is the user-determined upper bound of the number of nodes. The general form of a Cartesian genetic program is shown in Fig. 1. If the problem requires n_o program outputs, then n_o integers are generally added to the end of the genotype. In general, there may be a number of output genes (O_i) which specify where the program outputs are taken from. Each of these is an address of a node where the program output data is taken from. Nodes in columns cannot be connected to each other. In many cases graphs encoded are directed and feed-forward; this means that a node may only have its inputs connected to either input data or the output of a node in a previous column. The structure of the genotype is seen below the schematic in Fig. 1. All node function genes f_i are integer addresses in a look-up table of functions. All connection genes

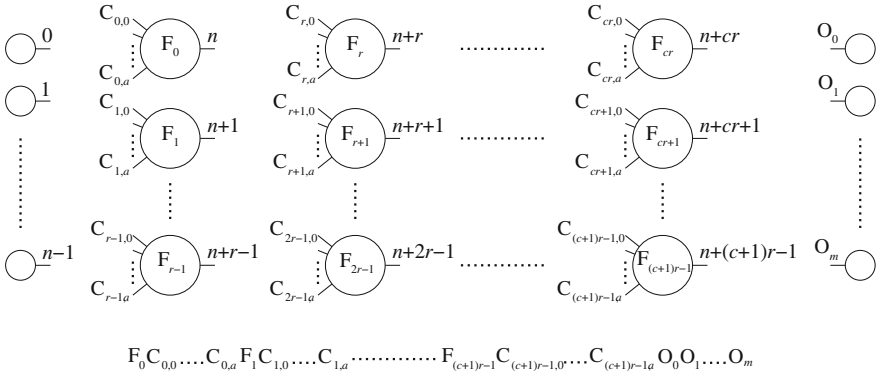

Fig. 1 General form of CGP. It is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c columns and n_r rows. The number of program inputs is n_i and the number of program outputs is n_o. Each node is assumed to take as many inputs as the maximum function arity a. Every data input and node output is labeled consecutively (starting at 0), which gives it a unique data address which specifies where the input data or node output value can be accessed (shown in the *figure* on the outputs of inputs and nodes)

C_{ij} are data addresses and are integers taking values between 0 and the address of the node at the bottom of the previous column of nodes.

CGP programs which solve computational problems are found using a search algorithm. The algorithm typically used is a simple kind of probabilistic hill-climber, known as a $1 + \lambda$ evolutionary algorithm [46]. Usually λ is chosen to be 4. This has the form shown in Algorithm 1.

Algorithm 1 The $(1 + 4)$ evolutionary strategy

1: **for all** i such that $0 \leq i < 5$ **do**
2: Randomly generate individual i
3: **end for**
4: Select the fittest individual, which is promoted as the parent
5: **while** a solution is not found **or** the generation limit is not reached **do**
6: **for all** i such that $0 \leq i < 4$ **do**
7: Mutate the parent to generate offspring i
8: **end for**
9: Generate the fittest individual using the following rules:
10: **if** an offspring genotype has a better or *equal* fitness than the parent **then**
11: Offspring genotype is chosen as fittest
12: **else**
13: The parent chromosome remains the fittest
14: **end if**
15: **end while**

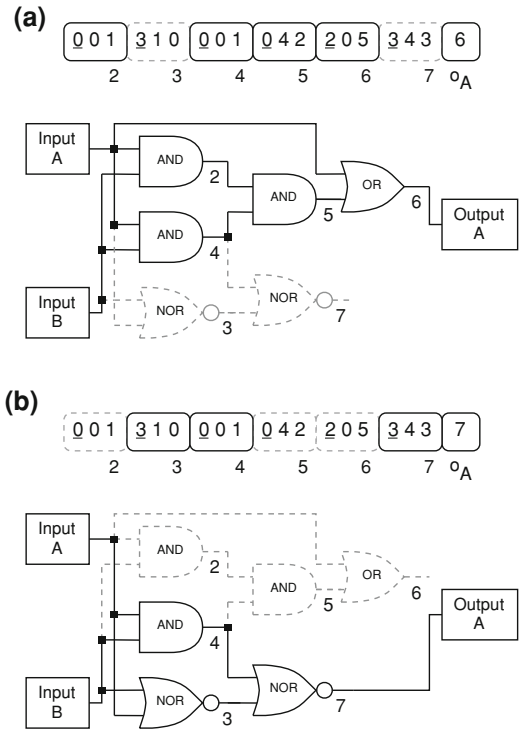

Fig. 2 An example of the point mutation operator before and after it is applied to a CGP genotype, and the corresponding phenotypes. A single point mutation occurs in the program output gene (o_A), changing the value from 6 to 7. This causes nodes 3 and 7 to become active, whilst making nodes 2, 5 and 6 inactive. The inactive areas are shown in *grey dashes*. **a** Before mutation, **b** After mutation

The mutation operator used in CGP is a point mutation operator. In a point mutation, an allele at a randomly chosen gene location is changed to another valid random value (see [45] for details). If a function gene is chosen for mutation, then a valid value is the address of any function in the function set. In cases where are is only one function allowed (e.g. all functions are sigmoidal neuron) then function genes are not required. If an input gene is chosen for mutation, then a valid value is the address of the output of any previous node in the genotype or of any program input. Also, a valid value for a program output gene is the address of the output of any node in the genotype or the address of a program input. The number of genes in the genotype that can be mutated in a single application of the mutation operator is defined by the user, and is normally a percentage of the total number of genes in the genotype.

When CGP programs are evolved, the connectivity, functionality and topology of the encoded graphical structures can change dramatically from generation to generation. The evolutionary algorithm is optimizing all these aspects simultaneously.

An example showing the application of a point mutation operator is shown in Fig. 2. In this example, the function nodes are all Boolean logic functions (with two inputs). The example also highlights how a small change in the genotype can sometimes produce a large change in the phenotype.

On line 10 of the procedure there is an extra condition: that when an offspring genotype in the population has the same fitness as the parent and there is no other offspring that is better than the parent, in that case the *offspring* is chosen as the new parent. This is a very important feature of the algorithm, which allows genotypes to change even when the phenotype does not. Such genotypes have mutations in genetic code that is inactive. Such inactive genes therefore have a neutral effect on genotype fitness. CGP genotypes are dominated by redundant genes. For instance, Miller and Smith showed that in genotypes having 4,000 nodes, the percentage of inactive nodes is approximately 95 %! [47]. The influence of neutrality in CGP has been investigated in detail [33, 47–49] and has been shown to be extremely beneficial to the efficiency of the evolutionary process on a range of test problems. The neutral drift of genotypes allows mutation to create many innovative variants of the current best genotype, some of which will occasionally contain phenotypes which are fitter than the parent.

3 A Neurocentric Model: The CGP Developmental Network

The CGPDN model has idealized the behaviour of a neuron in terms of seven main processes. The reasons for this have been discussed in more detail in [12, 28].

1. Local interaction among neighbouring branches of the same dendrite.
2. Processing of signals received from dendrites at the soma and deciding whether to fire an action potential.
3. Synaptic connections which transfer potential through axon branches to the neighbouring dendrite branches.
4. Dendrite branch growth and shrinkage. Production of new dendrite branches, removal of old branches.
5. Axon branch growth and shrinkage. Production of new axon branches, removal of old branches.
6. Creation or destruction of neurons.
7. Updating the synaptic weights (and consequently the capability to make synaptic connections) between axon branches and neighbouring dendrite branches.

Each aspect is incorporated with a separate chromosome (CGP program). The advantage of having a compartmentalized model is that different aspects of the model can be examined separately. Their utility to the whole can be assessed and if necessary the different compartments of the model can be refined.

In the CGPDN, neurons are placed randomly in a two dimensional grid so that they are only aware of their spatial neighbours (see Fig. 3). Each neuron is initially allocated a random number of dendrites, dendrite branches, one axon and a random number of axon branches. An integer variable that mimics electrical potential is used for internal computation in neurons and communication between neurons. Neurons receive information through dendrite branches, which is processed by the evolved

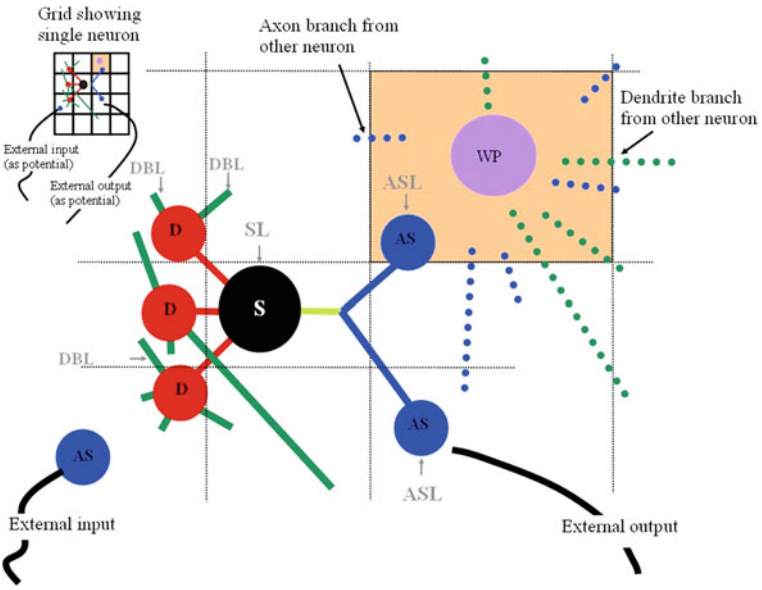

Fig. 3 On the *top left* a grid is shown containing a single neuron. The rest of the *figure* is an exploded view of the neuron. The neuron consists of seven evolved computational functions. Three are 'electrical' and process a simulated potential in the dendrite (D), soma (S) and axo-synapse branch (AS). Three more are developmental in nature and are responsible for the 'life cycle' of neural components (shown in *grey*). They decide whether dendrite branches (DBL), soma (SL) and axo-synaptic branches (ASL) should die, change or replicate. The remaining evolved computational function (WP) adjusts synaptic and dendritic weights and is used to decide the transfer of potential from a firing neuron to a neighbouring neuron

dendrite program (D) and transferred to the evolved soma program (S). S determines the final potential in the soma, which is compared to a threshold to determine whether it should fire or not. Axon branches transfer information only to dendrite branches in their proximity by passing the signals from all the neighbouring branches through a CGP program (AS), acting as an electrochemical synapse, which in turn updates the values of potential only in neighbouring branches. The weight processing chromosome (WP) adjusts the weights of potential connections to the synapse. The signal is transferred to the postsynaptic neuron having the largest weight. External inputs and outputs are also converted into potentials before being applied to the network.

3.1 Internal Neuron Variables

Four integer variables are incorporated into the CGPDN, representing either the fundamental properties of the neurons (*health*, *resistance*, and *weight*) or as an aid to computational efficiency (*state-factor*).

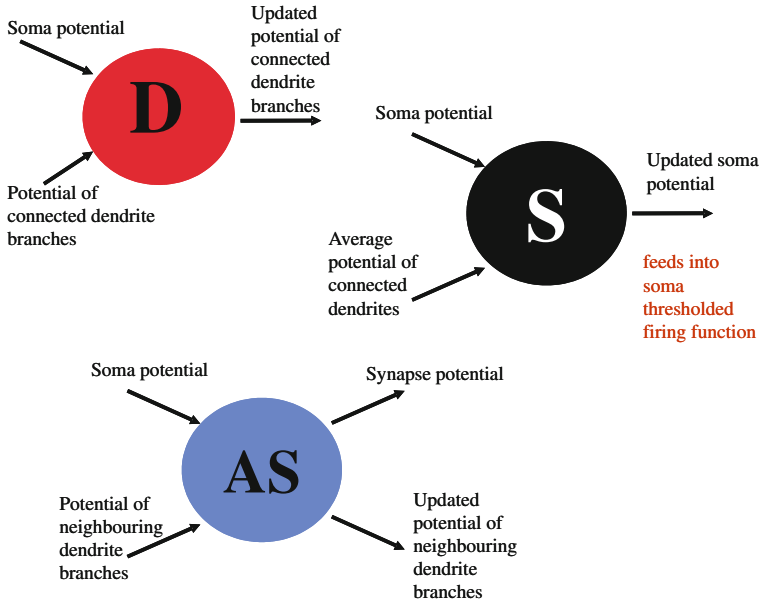

Fig. 4 Electrical processing in a neuron, showing the CGP programs for a dendrite branch, the soma and an axo-synaptic branch with their corresponding inputs and outputs

Associated with each dendrite branch and axo-synaptic connection are the variables *health*, *resistance* and *weight*. The values of these variables are adjusted by the CGP programs (see below). The *health* variable is used to govern the replication and/or death of dendrites and axon branches. A large value implicates replication a low value implicates removal (death). The *resistance* variable controls the growth and/or shrinkage of dendrites and axon branches. The weight variable is used in calculating the potentials in the network. Each soma has only two variables: *health* and *weight*.

The variable *state-factor* is used as a parameter to reduce the computational burden by keeping some of the neurons and branches inactive for a number of cycles. When the *state-factor* is zero, the neurons and branches are considered to be active and their corresponding program is run. The value of *state-factor* is affected by the CGP programs, as it is dependent on the outputs of the CGP electrical-processing chromosomes.

3.2 Electrical Processing

The electrical-processing chromosomes (D, S and AS) are responsible for signal processing inside neurons and communication between neurons. The inputs supplied to the CGP programs are shown in Fig. 4.

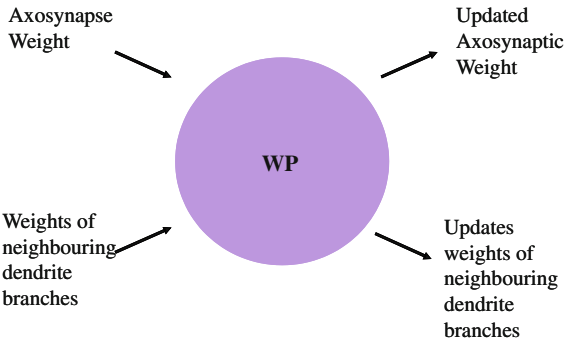

Fig. 5 Weight processing in an axo-synaptic branch, with its corresponding inputs and outputs

3.3 Weight Processing

The weight processing program (WP) is responsible for updating the *weights* of branches. The *weights* of axon and dendrite branches are also used to modulate and transfer the simulated potential [28].

Figure 5 shows the inputs and outputs to the weight-processing chromosome. The CGP program encoded in this chromosome takes as input the *weight* of the axo-synapse and the *neighbouring* dendrite branches of other neurons and produces their updated values as output. The synaptic potential produced at the axo-synapse is transferred to the dendrite branch having the highest weight after weight processing.

3.4 Developmental Aspects of Neurons

The DBL, ABL and SL CGP chromosomes (see Fig. 3) are responsible for increases or decreases in the numbers of neurons and neurite branches and also the growth and migration of neurite branches. The inputs and outputs of the programs encoded in these chromosomes are shown in Fig. 6.

3.5 Inputs and Outputs

The inputs are applied to the CGPDN through axon branches by using axo-synaptic electrical-processing chromosomes. The axon branches are distributed across the network in a similar way to the axon branches of neurons as shown in Fig. 7. These branches can be regarded as 'input neurons'. They take an input from the environment and transfer it directly to the axo-synapse input. When inputs are applied to the system, the program encoded in the axo-synaptic electrical branch chromosome is executed and the resulting signal is transferred to its neighbouring active dendrite branches.

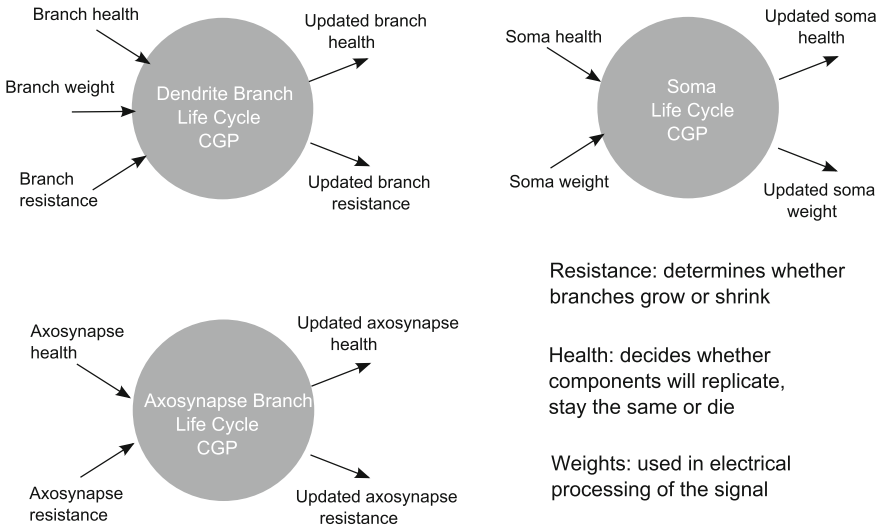

Fig. 6 Life cycle of neuron, showing CGP programs for life cycles in a dendrite branch, the soma and an axo-synapse branch, with their corresponding inputs and outputs

Similarly, there are output neurons which read the signal from the network through output dendrite branches. These output dendrite branches are distributed across the network as shown in Fig. 7. The branches are updated by the axo-synaptic chromosomes of the neurons in the same way as for other dendrite branches. The output from the output neuron is taken without further processing after every five cycles. The number of inputs and outputs can change at run time (during development), a new input or output branch can be introduced into the network, or an existing branch can be removed. This allows CGPDN to handle arbitrary numbers of inputs and outputs at run time.

The number of programs, that are run and transfer the potential from all active neurons to other active neurons is dependent on the number of active neural electrical components. Developmental programs determine the morphology of the neural network (i.e. the number of dendrite branches, axo-synapes and somae and how they are connected). The number of dendrites on each neuron is fixed, however the number of dendrite branches on each dendrite is variable and is determined by whether the developmental dendrite branch programs (DBL) in the past decided to eliminate or grow new branches. Every neuron is invested with a single axon, however, the number of axo-synapses attached to each axon is determined by whether the axo-synaptic branch program (ASL) in the past decided to grow or eliminate new axo-synapses. The number of neurons (initialized as a small randomly chosen number) is determined over time by whether soma developmental programs (SL) decided to replicate neurons or not (Fig. 8).

Whatever the number of programs that are run in the developing neural network, the size of the genotype is fixed and depends only on the sizes of the seven

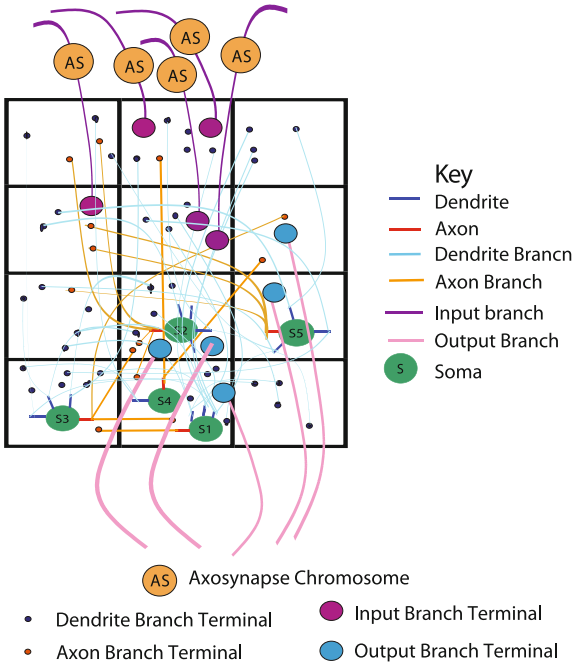

Fig. 7 Schematic illustration of a CGPDN defined over a 3×4 grid. The grid contains five neurons; each neuron has a number of dendrites with dendrite branches, and an axon with axon branches. Inputs are applied at five random locations in the grid using input axo-synapse branches by running axo-synaptic CGP programs. Outputs are taken from five random locations through output dendrite branches. Each *grid square* represents one location; the branches and soma are shown *spaced* for clarity. Each branch location is represented by where it terminates. Every location can have an arbitrary number of neurons and branches; there is no upper limit

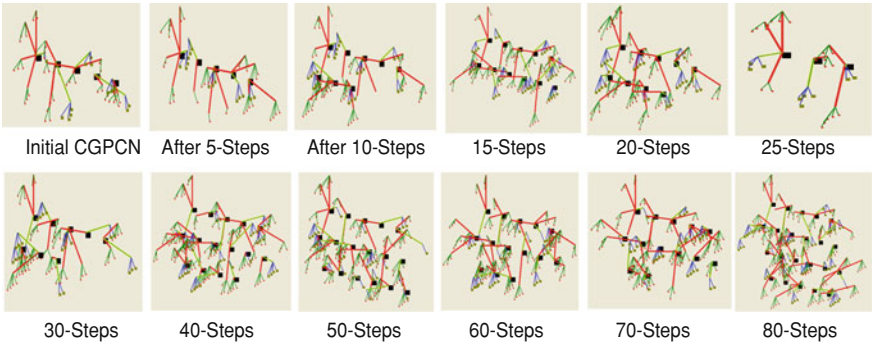

Fig. 8 Structural changes in a CGPDN network of a Wumpus World agent at different time steps. The network has five neurons at the start, and 21 neurons after completing 80 steps. *Black squares* are somae, *thick lines* (*red*) are dendrites, *yellowish green lines* are axons, *green lines* are dendrite branches, and *blue lines* show axon branches. Inputs and outputs are not shown

chromosomes that give rise to a network. This is one of the advantages of the developmental approach. A relatively simple collection of evolved programs can define an entire network of arbitrary complexity. The CGPDN model has been evaluated on a number of problems in artificial intelligence. Wumpus world [23], checkers [22, 25, 26] and maze solving [50]. Results show that the CGPDN produces networks that learn with experience (without further evolution). With structurally *different* networks they can recognize situations that have occurred before and cause the same actions. For instance, we observed that in a series of games of checkers, CGPDN players make appropriate, and often the same move, when a new game starts even though the neural network is different from the network that existed at the start of a previous game.

4 Self-Modifying CGP

Self-modifying Cartesian Genetic Programming (SMCGP) is a form of Genetic Programming founded on Cartesian Genetic Programming that is developmental in nature. In addition to the usual computational functions, it includes functions that can modify the program encoded in the genotype. This means that programs can be iterated to produce an infinite sequence of programs (phenotypes) from a single evolved genotype. It also allows programs to acquire more inputs and produce more outputs during this iteration.

Algorithm 2 gives a high-level overview of the process of mapping a genotype to a phenotype in SMCGP. The first stage of the mapping is the modification of the genotype. This happens through the use of evolutionary operators acting on the genotype. The developmental steps in the mapping are outlined in lines 3–8 of the algorithm. The first step is to make an exact copy of the genotype and call it the phenotype at iteration 0. After this, the self-modification operators are applied to produce the phenotype at the next iteration. Development stops when either a predefined iteration limit is achieved or it turns out that the phenotype has no self-modification operations that are active.

At each increment, the phenotype is evaluated and its fitness calculated. The underlying assumption here is that one is trying to solve a series of computational problems, rather than a single instance as is usual in GP. For instance, this might be a series of parity functions, ever-closer approximations to pi, or the natural numbers of the Fibonacci sequence. If the problem, however, has only a single instance (i.e. a classification problem), we can take a fixed number of iterations (either a user-defined parameter or evolved) and evaluate the single phenotype. Another possibility would be to iterate until no self-modification rules are active.

It is important to note that there are various ways in which there may be no active self-modification operations. Firstly, no self-modification operations may exist in the phenotype. Secondly, self-modification operations may be present but be non-coding. Thirdly, the self-modification operations may not be ‘activated’ when the

instructions encoded in the phenotype are executed. These various conditions will be discussed in the detailed description in the following sections.

Algorithm 2 Overview of genotype, phenotype and development

1: Generate genotype
2: Copy genotype to phenotype. Iteration, $i = 0$
3: **repeat**
4: Apply self-modification operations to phenotype i
5: increment i
6: Calculate fitness increment, f_i
7: **until** ((i equals number of iterations required) **OR** (No self-modification functions to do))
8: Evaluate phenotype fitness F from fitness increments, f_i

5 The Relation of SMCGP to CGP

The genetic representation in SMCGP has much in common with the representation used in CGP. The genotype encodes a graph (usually acyclic) that includes a list of node functions used and their connections. The arity of all functions is chosen to be equal to that of the largest-arity function in the function set. So, as in CGP, functions of lower arity ignore extraneous inputs. Although the form of SMCGP described here represents genotypes using a linear string of nodes, a two-dimensional form has been recently proposed [51].

5.1 Self-Modification Functions

The most significant difference between SMCGP and CGP is the addition of self-modification (SM) functions. These functions can be used in the genotype in the same manner as the more conventional computational operators, but at run time they provide different functionality. When the SMCGP phenotype is run, the nodes in the graph are parsed in a similar way to CGP. The graph is executed recursively by following nodes from the output nodes to the terminals (inputs). When computational functions are called, then—as usual—they operate on the data coming into the node.

When an SM node is called, the process is as follows. If an SM node is 'activated'; then its self-modification instructions are added to a list of pending manipulations which is called the *To-Do* list. The modifications in this list are then performed between iterations. In some implementations of SMCGP SM nodes are 'activated' if some numerical condition of its input data is obeyed (i.e. the first input is larger than the second). This makes the genotype-phenotype mapping data (and therefore context) dependent. Such a concept could be useful in ANNs (see later) as SM

Table 1 Examples of self-modification functions

Function name	Description
Delete (DEL)	Delete the nodes between $(P_0 + x)$ and $(P_1 + x)$
Add (ADD)	Add P_1 new random nodes after $(P_0 + x)$.
Move (MOV)	Move the nodes between $(P_0 + x)$ and $(P_1 + x)$ and insert after $(P_2 + x)$
Overwrite (OVR)	Copy the nodes between $(P_0 + x)$ and $(P_1 + x)$ to position $(P_2 + x)$, replacing existing nodes
Duplication (DUP)	Copy the nodes between $(P_0 + x)$ and $(P_1 + x)$ and insert after $(P_2 + x)$
Change connection (CHC)	Change the $(P_1 \bmod 3)$th connection of node P_0 to P_2
Change function (CHF)	Change the function of node P_0 to the function associated with P_1
Change argument (CHA)	Change the $(P_1 \bmod 3)$th argument of node P_0 to P_2

P_i are the evolved arguments of the self-modification functions; x represents the absolute position of the node in the graph, where the leftmost node has position 0. All additions are taken modulo (the number of nodes in the phenotype), this ensures they are always valid

operations could be activated, say, when the signal supplied to it was above a certain threshold (rather like the firing behaviour).

Many SM operators are imaginable, and Table 1 lists a few examples. In the table, we can see that the operators also require arguments. These come from the genotype and are described in Sect. 5.3. It is also worth noting that the indices for SM operations are defined relative to the current node.

5.2 Computational Functions

The computational functions used in SMCGP are typical of such functions in GP in general. They may be arithmetic operations such as addition, subtraction, multiplication, and division, or they may be mathematical functions such as *sin*, *exp* etc. In neural models one might choose sigmoid or hyperbolic tangent functions.

5.3 Arguments

Each node in the SMCGP genotype contains three integers defined in the range 0 to the maximum number of nodes in the genotype.[2] These numbers are evolved and are used in several ways by the SMCGP phenotype. The SM functions require several arguments to specify how graph modifications are to be carried out (see Table 1). These arguments are integers, and their values are calculated from the node's arguments.

During iteration of the genotype, the arguments can be altered by the SM function CHP ('change parameter'). This, in principle, allows storing of the state (i.e. a memory), since a phenotype could pass information to the phenotype at the next iteration through a collection of constant values.

[2] other reported implementations of SMCGP use floating point numbers.

5.4 Relative Addressing

The SM operators' ability to move, delete and duplicate sections of the graph means that the classical CGP approach of labelling nodes becomes cumbersome. Classical CGP uses *absolute addressing*, so that each node has an address and nodes reference each other using these addresses (this is what connection genes are—see Sect. 2).

To simplify the representation, absolute addresses were replaced with relative addresses. Now, instead of a node containing an absolute address of another node, it specifies how many nodes back from its position are required to make a connection. The connections in the genotype are now defined as positive integers that are greater than 0 (which prevents cycles).

When the graph is run, the interpreter can calculate where a node gets its input values from by just subtracting the connection value from the current address of the node. If the node addresses a value that is not in the graph (i.e. connects too far back), then a default value is returned (in the case of numeric applications this is 0).

The arguments of SM operators are also defined relative to the current node (see Table 1). The relative addressing allows subgraphs to be placed or duplicated in the graph whilst retaining their semantic validity. This means that subgraphs could represent the same subfunction, but act on different inputs. This can be done without recalculating any node addresses, thus maintaining validity of the whole graph. So subgraphs can be used as functions in the sense of the ADFs of standard GP.

5.5 Input and Output Nodes

Most, if not all, genetic programming implementations have a fixed number of inputs. This certainly makes sense when there is a constant or bounded number of inputs over the lifetime of a program. However, it prevents the program from scaling to larger problem sizes by increasing the number of inputs it uses—and this in turn may prevent general solutions from being found. Similarly, most GP systems have a fixed number of outputs. In tree-based GP, there is typically a single output. In classical CGP, a number of input nodes are placed at one end of the graph, and these are used as the starting point for the recursive interpretation of the program. To allow an arbitrary number of inputs and outputs, SMCGP introduces several new functions into the basic function set. These are shown in Table 2.

The interpreter now keeps track of an input pointer, which points to a particular input in the array of input values. Calling the function INPI returns the value that the pointer is currently on, and then moves the pointer to the next value. When the pointer runs out of inputs, it resets to the first input. Similarly, the DECI function returns an input but then moves the pointer to the previous value. Sometimes it may not be convenient or useful to move by only one input at a time, hence the SKPI is included. It moves the pointer a number of places. The number is arrived at by adding (modulo the number of program inputs) P_0 to the input pointer and then

Table 2 Input and output functions

Function	Operation
INCI	Return input pointed to by current_input, increment current_input
DECI	Return input pointed to by current_input, decrement current_input
SKPI	Return input pointed to by current_input, current_input = current_input + P_0
INCO	write data to current_output element of output_register, increment current_output
DECO	write data to current_output element of output_register, decrement current_output
SKPO	write data to current_output element of output_register, current_output = current_output + P_0

P_0 is the first argument gene; current_input wraps around so that when current_input equals the number of program inputs, current_input is set to zero; current_output also wraps around and writes outputs to an output register that has a number of elements equal to the number of program outputs. The output register is initialized with zeros

returning that input. The output functions work in a similar manner except that they write to an output register which has the same number of elements as the desired program outputs at that iteration. The register is filled with defaults values of zero. By duplicating any input and output functions the SMCGP phenotypes can acquire more inputs and outputs when they are iterated.

6 SMCGP Performance and Applications

The performance of SMCGP, in terms of the average number of genotype evaluations required to solve a problem, has been evaluated on a number of problems.[3] It has also been compared with other published results of other approaches [39] and also compared with standard CGP and a form of CGP with ADFS [44]. In all cases SMCGP solves easy instances of problems in a larger number of evaluations than other approaches, however it scales much better so that it solves harder instances in considerably less evaluations. Indeed it has also been shown to be able to produce mathematically provable general solutions to some problems. This has been done for parity, binary addition, and computing pi and e. Clearly SMCGP is a powerful and flexible evolutionary technique. It is natural, therefore, to investigate its utility in the evolution of developmental neural networks. We discuss this idea in the next section.

[3] This work has not involved ANNs.

Table 3 Suggested self-modification functions for ANNs

Function name	Description
Add connection (ADDC)	Add a random connection to nodes $(P_0 + x)$ to $(P_1 + x)$
Remove connection (REMC)	Remove a connection at random for all nodes $(P_0 + x)$ to $(P_1 + x)$
Increase weight (INCW)	Increase weight by a fixed percentage for all nodes $(P_0 + x)$ to $(P_1 + x)$
Derease weight (DECW)	Decrease weight by a fixed percentage for all nodes $(P_0 + x)$ to $(P_1 + x)$

P_i are the evolved arguments of the self-modification functions; x represents the absolute position of the node in the graph, where the leftmost node has position 0. All additions are taken modulo (the number of nodes in the phenotype), this ensures they are always valid

7 A Holocentric Model: SMCGP Artificial Neural Networks

It is relatively straightforward to adapt the SMCGP approach so that it constructs developmental neural networks. Here the function set can be a collection of conventionally used neuron functions (i.e. sigmoidal, hyperbolic tangent). New SM functions, in addition to those described in Table 1 need to be designed that are specific to ANNs. Suggested SM functions relevant to ANNs are shown in Table 3. With the complete set of functions (and assuming say a sigmoid computational node function) shown in Tables 1 and 3 one would have genotypes which were valid ANNs, that when iterated could produce ANNs with arbitrary topology and weight adjustment. The topology of the network would change through the action of embedded self-modification instructions (see Tables 1 and 2). Also weights at each iteration could be changed in the network through the action of embedded weight altering self-modification functions (see Table 3). This combined with the incremental fitness function described in Algorithm 2 could allow conventional ANNs to develop their own topology while performing a task. An additional attractive feature of the SMCGP approach to ANNs is that it should be relatively easy to adopt other models of artificial neurons, particularly spiking neural networks [52].

7.1 Process of Learning in SMCGP Artificial Neural Networks

The process of learning in SMCGP artificial neural networks would happen through a combination of topological changes (numbers of neurons and connections) and synaptic changes. It is important to note that evolved genotype would be simultanenously both (a) a valid ANN and (b) a set of rules that would produce an arbitrary sequence of ANNs. It would be a valid ANN because all computational nodes would be a standard neuron function (e.g. sigmoid) and all connections would have explicit weights in the genotype. However, on iteration (i.e. application of the embedded self-modification instructions) a new network would be produced. It may

turn out to be necessary to apply weight adjustment self-modification instructions more frequently than topological change inducing instructions. This could be done via a weight-adjustment algorithm (i.e. backpropagation or a search algorithm) operating within a single iteration (in the sense of Algorithm 2). However, by supplying a series of learning problems of different types, it should be possible to evolve an algorithm that when iterated, adjusts weights and makes topological changes in such a way that effective learning takes place. For instance, a number of the fitness increments referred to in Algorithm 2 could be acrued during training on a particular problem, before iterating the SMCGP phenotype on another learning problem. The number of iterations that are devoted to a particular learning problem would be a choice for the experimenter. In principle the manner in which fitness increments are calculated on a particular learning problem could be similar to the way fitness is evaluated in neuroevolutionary approaches to artificial neural networks. There are many issues here that remain for future research.

8 Neuro-Centric Versus Holocentric ANNs

Perhaps the greatest problem with neuro-centric ANNs is the complexity of the neurons. In the approach outlined above the neurons contained seven evolved chromosomes. The action of these chromosomes was to alter internal variables that resulted in electrical and morphological changes in the neurons. The values of these variables were used to determine increases in neural sub-component weights, neurite topology, transfer of potential, neuron and neurite replication and death and so on. To do this required a large number of additional rules many of which could be important for the success of the technique. The scientific study of such complex models is consequently difficult. If the model proves to be successful, what rules and assumptions are essential to that success? Though it is true that neuro-centric models can benefit from improvements in our understanding of neuroscience, this inevitably compounds the computational complexity problem. Science has not hitherto attempted to build models of complex systems using extremely complex primitives (i.e. a neuron), however it is clear that natural evolution has built a fundamental building block of life that is of staggering complexity. This is the cell. Only time will tell as to whether it is possible to extract from neuroscience a sufficiently simple and computationally efficient model of a neuron.

Holocentric models are therefore very attractive as they do not simulate many low level details of neurons. For instance, SMCGP can build a new computational network by carrying out simple graph altering operations on the genotype. HyperNEAT can generate whole neural networks by running a program that is a function of four variables. This makes them computationally more efficient. The drawback that these models have is that informing them from neuroscience is much more difficult. Holocentric models are much more abstract than neuro-centric models and are founded on high level assumptions. One of these was historically, the idea that synaptic weights were sufficient for general learning. The danger with making such

high-level assumptions is that they may produce models that have hitherto unknown limitations. Despite this it appears that abstracting developmental aspects of neuroscience and informing holocentric models with this appears to be a very promising direction.

9 Conclusions and Future Outlook

In this chapter we have outlined two approaches to developmental neural networks. One is neuro-centric and evolves complex computational models of neurons, the other is holocentric and use evolution and development at a whole network level. Both approaches have merits and potential drawbacks. The main issue with neuro-centric models is related to keeping the model complexity to be as small as possible and making the model computationally efficient. Holocentric models are simpler and more efficient but make high level assumptions that may ultimately restrict their power and generality. Both approaches are worthy of continued attention.

It is our view that one of the main aims of the neural networks should be to produce networks that are capable of general learning. General learning refers to an ability to learn in multiple task domains without the occurrence of interference. A fundamental problem in creating general learning systems is the *encoding problem*. This is where the data that is fed into the neural networks has to be specifically encoded for each problem. Biological brains avoid this by using sensors to acquire information about the world and actuators to change it. We suggest that such universal representations will be required in order for developmental artificial neural networks to show general learning. Thus we feel that general learning systems can only be arrived at through systems that utilize sensory data from the world. Essentially this means that such systems need to be implemented on physical robots. This gives us an even greater incentive to construct highly efficient models of neural networks.

References

1. W.S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophy. **5**, 115–133 (1943)
2. E.R. Kandel, J.H. Schwartz, T.M. Jessell, *Principles of Neural Science*, 4th edn. (McGraw-Hill, New York, 2000)
3. R.M. French, Catastrophic forgetting in connectionist networks: causes, consequences and solutions. Trends Cogn. Sci. **3**(4), 128–135 (1999)
4. M. McCloskey, N.J. Cohen, Catastrophic interference in connectionist networks: the sequential learning problem. Psychol. Learn. Motiv. **24**, 109–165 (1989)
5. R. Ratcliff, Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. Psychol. Rev. **97**, 285–308 (1990)
6. S. Judd, On the complexity of loading shallow neural networks. J. Complex. **4**, 177–192 (1988)
7. E.B. Baum, A proposal for more powerful learning algorithms. Neural Comput. **1**, 201–207 (1989)

8. S.E. Fahlman, C. Lebiere, *The cascade-correlation architecture*, ed. by D.S. Touretzky. Advances in Neural Information Processing Systems (Morgan Kaufmann, San Mateo, 1990)
9. M. Frean, The upstart algorithm: a method for constructing and training feedforward neural networks. Neural Comput. **2**, 198–209 (1990)
10. P.T. Quinlan, Structural change and development in real and artificial networks. Neural Netw. **11**, 577–599 (1998)
11. P.T. Quinlan (ed.), *Connectionist Models of Development* (Psychology Press, New York, 2003)
12. J.F. Miller, G.M. Khan, Where is the brain inside the brain? on why artificial neural networks should be developmental. Memet. Comput. **3**(3), 217–228 (2011)
13. J.R. Smythies, *The Dynamic Neuron* (MIT Press, Cambridge, 2002)
14. F. Valverde, Rate and extent of recovery from dark rearing in the visual cortex of the mouse. Brain Res. **33**, 1–11 (1971)
15. J.A. Kleim, E. Lussnig, E.R. Schwartz, T.A. Comery, W.T. Greenough, Synaptogenesis and fos expression in the motor cortex of the adult rat after motor skill learning. J. Neurosci **16**, 4529–4535 (1996)
16. J.A. Kleim, K. Vij, D.H. Ballard, W.T. Greenough, Learning-dependent synaptic modifications in the cerebellar cortex of the adult rat persist for at least four weeks. J. Neurosci **17**, 717–721 (1997)
17. M.L. Mustroph, S. Chen, S.C. Desai, E.B. Cay, E.K. Deyoung, J.S. Rhodes. Aerobic exercise is the critical variable in an enriched environment that increases hippocampal neurogenesis and water maze learning in male C57BL/6J mice. Neuroscience (2012), Epub ahead of print
18. A.D. Tramontin, E. Brenowitz, Seasonal plasticity in the adult brain. Trends Neurosci. **23**, 251–258 (2000)
19. E.A. Maguire, D.G. Gadian, I.S. Johnsrude, C.D. Good, J. Ashburner, R.S.J. Frackowiak, C.D. Frith, Navigation-related structural change in the hippocampi of taxi drivers. PNAS **97**, 4398–4403 (2000)
20. S. Rose, *The Making of Memory: From Molecules to Mind* (Vintage, London, 2003)
21. A.S. Dekaban, D. Sadowsky, Changes in brain weights during the span of human life. Ann. Neurol. **4**, 345–356 (1978)
22. G.M. Khan, J.F. Miller, *Evolution of cartesian genetic programs capable of learning*, ed. by F. Rothlauf. Conference on Genetic and Evolutionary Computation (GECCO) (ACM, 2009) pp. 707–714
23. G.M. Khan, J.F. Miller, D.M. Halliday, *Coevolution of intelligent agents using Cartesian genetic programming*. Conference on Genetic and Evolutionary Computation (GECCO) (2007), pp. 269–276
24. G.M. Khan, J.F. Miller, D.M. Halliday, *Breaking the synaptic dogma: Evolving a neuro-inspired developmental network*, ed. by X. Li, M. Kirley, M. Zhang, D.G. Green, V. Ciesielski, H.A. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K.C. Tan, J. Branke, Y. Shi. Simulated Evolution and Learning, 7th International Conference, SEAL 2008, Melbourne, Australia, December 7–10, 2008. Proceedings, volume 5361 of Lecture Notes in Computer Science (Springer, 2008), pp. 11–20
25. G.M. Khan, J.F. Miller, D.M. Halliday, *Coevolution of neuro-developmental programs that play checkers*, ed. by G. Hornby, L. Sekanina, P.C. Haddow. Evolvable Systems: From Biology to Hardware, 8th International Conference, ICES 2008, Prague, Czech Republic, September 21–24, 2008. Proceedings, volume 5216 of Lecture Notes in Computer Science (Springer, 2008), pp. 352–361
26. G.M. Khan, J.F. Miller, D.M. Halliday, *Developing neural structure of two agents that play checkers using cartesian genetic programming*, ed. by C. Ryan, M. Keijzer. Conference on Genetic and Evolutionary Computation (GECCO) Companion Material (ACM, 2008), pp. 2169–2174
27. G.M. Khan, J.F. Miller, D.M. Halliday, In search of intelligent genes: the cartesian genetic programming computational neuron (cgpcn), in *Proceedings of the IEEE Congress on Evolutionary Computation*, CEC 2009, Trondheim, Norway, 18–21 May, 2009 (IEEE, 2009), pp. 574–581

28. G.M. Khan, J.F. Miller, D.M. Halliday, Evolution of cartesian genetic programs for development of learning neural architecture. Evol. Comput. **19**(3), 469–523 (2011)
29. K.O. Stanley, D.B. D'Ambrosio, J. Gauci, A hypercube-based encoding for evolving large-scale neural networks. Artif. Life **15**, 185–212 (2009)
30. F. Gruau, Automatic definition of modular neural networks. Adapt. Behav. **3**, 151–183 (1994)
31. J.R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection* (MIT Press, Cambridge, 1992)
32. J.F. Miller, An *Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach*. Conference on Genetic and Evolutionary Computation (GECCO) (Morgan Kaufmann, 1999), pp. 1135–1142
33. J.F. Miller, P. Thomson, Cartesian Genetic Programming, in *Proceedings of the European Conference on Genetic Programming*, vol. 1802 of LNCS (Springer, 2000), pp. 121–132
34. S. Harding, J.F. Miller, W. Banzhaf, *A survey of self modifying CGP*, ed. by R. Riolo, T. McConaghy, E. Vladislavleda. Genetic Programming Theory and Practice VIII, 2010 (Springer, 2010) pp. 91–107
35. S. Harding, J.F. Miller, W. Banzhaf, Self-modifying Cartesian Genetic Programming, in *Proceedings of the Genetic and Evolutionary Computation Conference* (2007), pp. 1021–1028
36. S. Harding, J.F. Miller, W. Banzhaf, *Evolution, development and learning using self-modifying cartesian genetic programming*, ed. by F. Rothlauf. Conference on Genetic and Evolutionary Computation (GECCO) (ACM, 2009), pp. 699–706
37. S. Harding, J.F. Miller, W. Banzhaf, *Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing*, ed. by L. Vanneschi, S. Gustafson, A. Moraglio, I. De Falco, M. Ebner. Genetic Programming, 12th European Conference, EuroGP 2009, Tübingen, Germany, April 15–17, 2009, Proceedings, volume 5481 of Lecture Notes in Computer Science (Springer, 2009), pp. 133–144
38. S. Harding, J.F. Miller, W. Banzhaf, Self modifying cartesian genetic programming: Parity, in *Proceedings of the IEEE Congress on Evolutionary Computation*, CEC 2009, Trondheim, Norway, 18–21 May, IEEE, 2009 (2009), pp. 285–292
39. S. Harding, J.F. Miller, W. Banzhaf, Developments in cartesian genetic programming: self-modifying cgp. Genet. Program. Evolvable Mach. **11**(3–4), 397–439 (2010)
40. S. Harding, J.F. Miller, W. Banzhaf, *Self modifying cartesian genetic programming: finding algorithms that calculate pi and e to arbitrary precision*, ed. by M. Pelikan, J. Branke. Conference on Genetic and Evolutionary Computation (GECCO) (ACM, 2010), pp. 579–586
41. M.M. Khan, G.M. Khan, J.F. Miller, Efficient representation of recurrent neural networks for Markovian/Non-Markovian non-linear control problems, ed. by A.E. Hassanien, A. Abraham, F. Marcelloni, H. Hagras, M. Antonelli, T.-P. Hong, in *Proceedings of the International Conference on Intelligent Systems Design and Applications* (IEEE, 2010), pp. 615–620
42. M.M. Khan, G.M. Khan, J.F. Miller, Evolution of neural networks using cartesian genetic programming, in *Proceedings of the IEEE Congress on Evolutionary Computation*, CEC 2010, Barcelona, Spain, 18–23 July 2010 (IEEE, 2010)
43. M.M Khan, G.M. Khan, J.F. Miller, Evolution of optimal anns for non-linear control problems using cartesian genetic programming, ed. by H.R. Arabnia, D. de la Fuente, E.B. Kozerenko, J.A. Olivas, R. Chang, P.M. LaMonica, R.A. Liuzzi, A.M.G. Solo, in *Proceedings of the 2010 International Conference on Artificial Intelligence*, ICAI 2010, July 12–15, 2010, Las Vegas Nevada, USA, vol. 2 (CSREA Press, 2010), pp. 339–346
44. J.A. Walker, J.F. Miller, The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Trans. Evolut. Comput. **12**(4), 397–417 (2008)
45. J.F. Miller (ed.), *Cartesian Genetic Programming*. Natural Computing Series (Springer, Berlin, 2011)
46. I. Rechenberg, Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Ph.D. thesis, Technical University of Berlin, Germany, (1971)
47. J.F. Miller, S.L. Smith, Redundancy and computational efficiency in cartesian genetic programming. IEEE Trans. Evolut. Comput. **10**(2), 167–174 (2006)

48. V.K. Vassilev, J.F. Miller, *The Advantages of Landscape Neutrality in Digital Circuit Evolution*. International Conference on Evolvable Systems, vol. 1801 of LNCS (Springer, 2000), pp. 252–263
49. T. Yu, J.F. Miller, Neutrality and the evolvability of Boolean function landscape, in *Proceedings of the European Conference on Genetic Programming*, vol. 2038 of LNCS (Springer, 2001), pp. 204–217
50. G.M. Khan, J.F Miller, Solving mazes using an artificial developmental neuron, in *Proceedings of the Conference on Artificial Life (ALIFE) XII* (MIT Press, 2010), pp. 241–248
51. S. Harding, J.F. Miller, W. Banzhaf, *SMCGP2: Self-modifying Cartesian Genetic Programming in Two Dimensions*. Conference on Genetic and Evolutionary Computation (GECCO) (ACM, 2011), pp. 1491–1498
52. W. Gerstner, W.M. Kistler, *Spiking Neuron Models* (Cambridge University Press, Cambridge, 2002)