Evolving developmental programs that build neural networks for solving multiple problems

Julian F. Miller, Dennis G. Wilson, and Sylvain Cussat-Blanc

Abstract A developmental model of an artificial neuron is presented. In this model, a pair of neural developmental programs develop an entire artificial neural network of arbitrary size. The pair of neural chromosomes are evolved using Cartesian Genetic Programming. During development, neurons and their connections can move, change, die or be created. We show that this two-chromosome genotype can be evolved to develop into a single neural network from which multiple conventional artificial neural networks can be extracted. The extracted conventional ANNs share some neurons across tasks. We have evaluated the performance of this method on three standard classification problems: cancer, diabetes and the glass datasets. The evolved pair of neuron programs can generate artificial neural networks that perform reasonably well on all three benchmark problems simultaneously. It appears to be the first attempt to solve multiple standard classification problems using a developmental approach.

1 Introduction

Artificial neural networks (ANNs) were first proposed seventy-five years ago [26] Yet, ANNs still have poorer general learning capabilities than relatively simple organisms. Organisms can learn to perform well on many tasks and can generalise from few examples. Most ANNs models encode learned knowledge solely in the

Julian F. Miller
University of York, Heslington, York, YO10 5DD, UK e-mail: julian.miller@york.ac.uk

Dennis G. Wilson
University of Toulouse, IRIT - CNRS - UMR5505, 21 allee de Brienne, Toulouse, France 31015
e-mail: dennis.wilson@irit.fr

Sylvain Cussat-Blanc
University of Toulouse, IRIT - CNRS - UMR5505, 21 allee de Brienne, Toulouse, France 31015
e-mail: sylvain.cussat-blanc@irit.fr

form of connection strengths (i.e. weights). Biological brains do not learn merely by the adjustment of weights, they undergo topological changes during learning. Indeed, restricting learning to weight adjustment leads to "catastrophic forgetting" (CF) in which ANNs trained to perform well on one problem, forget how to solve the original problem when they are re-trained on a new problem [9, 25, 34]. Although the original inspiration for ANNs came from knowledge about the brain, very few ANN models use evolution and development, both of which are fundamental to the construction of the brain [29]. In principle, developmental neural approaches could alleviate catastrophic forgetting in at least two ways. Firstly, new networks could form in response to learning. Secondly, by growing numerous connections between pairs of neurons. In this way the influence of individual weighted connection could be lessened.

Developmental neural networks have not widely been explored in the literature and there remains a need for concerted effort to explore a greater variety of effective models. In this paper, we propose a new conceptually simple neural model. We suggest that at least two neural programs are required to construct neural networks. One to represent the neuron soma and the other the dendrite. The role of the soma program is to allow neurons to move, change, die or replicate. For the dendrite, the program needs to be able to grow and change dendrites, cause them to die and also to replicate. Since developmental programs build networks that change over time it is necessary to define new problem classes that are suitable to evaluate such approaches. We argue that trying to solve multiple computational problems (potentially even of different types) is an appropriate class of problems.

In this chapter, we show that the pair of evolved programs can build a network from which multiple conventional ANNs can be extracted each of which can solve a different classification problem. As far as we can tell, this is the first work that attempts to evolve developmental neural networks that can solve multiple problems, indeed it appears to be the first attempt to solve standard classification problems using a developmental approach. We investigate many parameters and algorithmic variants and assess experimentally which aspects are most associated with good performance. Although we have concentrated in this paper on classification problems, our approach is quite general and it could be applied to a much wider variety of problems.

2 Related work

A number of authors have investigated ways of incorporating development to help construct ANNs [24] and [42]. Researchers have investigated a variety of genotype representations at different levels of abstraction. Cangelosi et al. defined genotypes which were a mixture of variables, parameters, and rules (e.g. cell type, axon length and cell division instructions) [4]. The task was to control a simple artificial organism. Rust et al constructed a genotype consisting of developmental parameters (encoded in binary) that controlled the times at which dendrites could branch and how

the growing tips would interact with patterns of attractants placed in an environment [38]. Balaam investigated controlling simulated agents using a two-dimensional area with chemical gradients in which neurons were either sensors, affectors, or processing neurons according to location [2]. The neurons were defined as standard CTRNNS. The genotype was effectively divided into seven chromosomes each of which read the concentrations of the two chemicals and the cell potential. Each chromosome provided respectively the neuron bias, time constant, energy, growth increment, growth direction, distance to grow and new connection weight.

Gruau used a more abstract approach, called cellular encoding in which ANNs were developed using graph grammars [12, 13]. He evaluated this approach on hexapod robot locomotion and pole-balancing. Kodjabachian and Meyer used a "geometry-orientated" variant of cellular encoding to develop recurrent neural networks to control the behaviour of simulated insects [23].

Jacobi presented a low-level approach in which cells used artificial genetic regulatory networks (GRNs). The GRN produced and consumed simulated proteins that defined various cell actions (protein diffusion movement, differentiation, division, threshold). After a cellular network had developed it was interpreted as a neural network [18]. Eggenberger also used an evolved GRN [6]. A neural network phenotype was obtained by comparing simulated chemicals in pairs of neurons to determine if the neurons are connected and whether the connection is excitory or inhibitory. Weights of connections were initially randomly assigned and Hebbian learning used to adjust them subsequently. Astor and Adami also encoded a form of GRN together with an artificial chemistry (AC), in which cells were predefined to exist in a hexagonal grid. Genes encoded conditions involving concentrations of simulated chemicals which determine the level of activation of cellular actions (e.g. grow axon or dendrite, increase or decrease weight, produce chemical) [1]. They evaluated the approach on a simple artificial organism.

Federici used a simple recursive neural network as a developmental cellular program [7]. In his model, cells could change type, replicate, release chemicals or die. The type and metabolic concentrations of simulated chemicals in a cell were used to specify the internal dynamics and synaptic properties of its corresponding neuron. The position of the cell within the organism is used to produce the topological properties of neuron: its connections to inputs, outputs and other neurons. From the cellular phenotype, Federici interpreted a network of spiking neurons to control a Khepera robot.

Some researchers have studied the potential of Lindenmeyer systems for developing artificial neural networks. Kitano used a kind of L-system in which he evolved matrix re-writing rules to develop an adjacency matrix defining a neural network [22]. Boers and Kuiper adapted L-systems to develop artificial feed-forward neural networks [3]. They found that this method produced more modular neural networks that performed better than networks with a predefined structure. They showed that their method could produce ANNs for solving problems such as the XOR function. Hornby and Pollack evolved L-systems to construct complex robot morphologies and neural controllers [16, 15].

Downing adopted a higher-level approach which avoided axonal and dendritic growth, while maintaining key aspects of cell signaling, competition and cooperation of neural topologies [5]. He applied this technique to the control of a multi-limbed starfish-like animat.

Khan and Miller created a complex developmental neural network model that evolved seven programs each representing various aspects of biological neurons [19]. These were divided into two categories. Three of the CGP encoded chromosomes were responsible for 'electrical' processing of the 'potentials'. These were the dendrite, soma and axo-synapse chromosomes. One chromosome was devoted to updating the weights of dendrites and axo-synapses. The remaining three chromosomes were developmental responsible for updating the neural variables for the soma (health and weight), dendrites (health, weight and length) and axo-synapse (health, length). The evolved developmental programs were responsible for the death and replication of neural components. The model was used in various applications: intelligent agent behaviour (wumpus world), checkers playing, and maze navigation [20, 21].

Stanley introduced the idea of using evolutionary algorithms to build neural networks constructively (called NEAT). The network is initialised as a simple structure, with no hidden neurons consisting of a feed-forward network of input and output neurons. An evolutionary algorithm controls the gradual complexification of the network by adding a neuron along an existing connection, or by adding a new connection between previously unconnected neurons [39]. However, using random processes to produce more complex networks is potentially very slow. It also lacks biological plausibility since natural evolution does not operate on aspects of the brain directly. Later Stanley introduced an interesting extension to the NEAT approach called HyperNEAT [41] which uses an evolved generative encoding called a Compositional Pattern Producing Network (CPPN) [40]. The CPPN takes coordinates of pairs of neurons and outputs a number which is interpreted as the weight of that connection. The advantage this brings is that ANNs can be evolved with complex patterns where collections of neurons have similar behaviour depending on their spatial location. It also means that one evolved function (the CPPN) can determine the strengths of connections of many neurons. It is a form of non-temporal development, where geometrical relationships are translated into weights.

Developmental Symbolic Encoding (DSE) [43] combines concepts from two earlier developmental encodings, Gruau's cellular encoding and L-systems. Like HyperNEAT it can specify connectivity of neurons via evolved geometric patterns. It was shown to outperform HyperNEAT on a shape recognition problem defined over small pixel arrays. It could also produce partly general solutions to a series of even-parity problems of various sizes. Huizinga et al. added an additional output to the CPP program in HyperNEAT that controlled whether or not a connection between a pair of neurons was expressed or not [17]. They showed that the new approach produced more modular solutions and superior performance to HyperNEAT on three specially devised modular problems.

Evolvable-substrate HyperNEAT (ES-HyperNEAT) implicitly defined the positions of the neurons [35], however it proved to be computationally expensive. Iter-

ated ES-HyperNEAT proposed a more efficient way to discover suitable positioning of neurons [37]. This idea was taken further leading to Adaptive HyperNEAT which demonstrated that not only could patterns of weights be evolved but also patterns of local neural learning rules [36]. Like [17] in Adaptive HyperNEAT Risi et al. increased the number of outputs from the CPPN program to encode learning rate and other neural parameters.

3 The neuron model

Our aim is to construct a *minimal* developmental model. Minimal means that if we take a snapshot of the neural network at a particular time we would see a conventional graphs of neurons, weighted connections and a standard activation functions. However, to make a *developmental* neural network we require a mechanism whereby the ANN can change over time (possibly even during training). In addition, we take a cellular view of development, in which an entire network is developed from a few cells (possibly a single cell). The network itself grows from the interaction of neurons acting in parallel (but sequentially simulated).

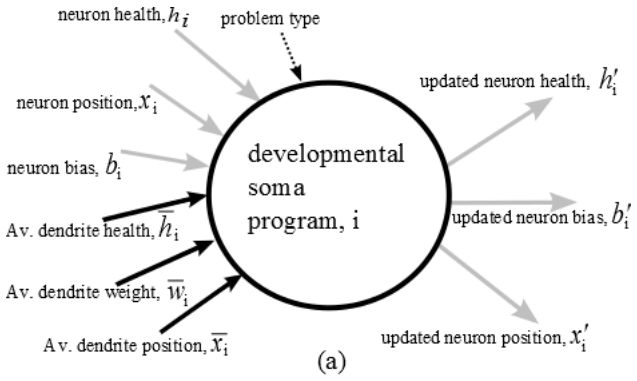
To construct such a developmental model of an artificial neural network we need neural programs that not only apply a weighted sum of inputs to an activation function to determine the output from the neuron, but a program that can adjust weights, create or delete connections, and create or delete neurons. Following [21] we have used the concept of health to make this possible.

The model is illustrated in in Fig. 1. The neural programs are represented using Cartesian Genetic Programming (CGP) (see Sect. 4). The programs are actually sets of mathematical equations that read variables associated with neurons and dendrites to output updates of those variables. This approach was inspired by some aspects of a developmental method for evolving graphs and circuits proposed by Miller and Thomson [32]. It was also influenced by some of the ideas described in [21]. In the proposed model, weights are determined from a program that is a function of neuron position, together with the health, weight and length of dendrites. It is neuro-centric and temporal in nature. Thus the neural networks can change over time.

The inputs to the soma program are as follows: the health, bias and position of the neuron and the average health, length and weight of all dendrites connected to the neuron and problem type.

The problem type is a constant (in range [-1, 1]) which indicates whether a neuron is not an output or in the case of an output neuron what computational problem the output neuron belongs to. Let P_t denote the computational problem. Define $P_t = 0$ to denote a non-output neuron, and P_t =1,2 or N_p to respectively denote output neurons belonging to different computational problems. Where, N_p denotes the number of computational problems. We define the problem type input to be given by $-1 + 2P_t/N_p$. For example, if the neuron is not an output neuron the problem type input is -1.0. If it is an output neuron belonging to the last problem its value is 1.0. For all other computational problems its value is a value greater than -1.0 and less

Fig. 1 The model of a developmental neuron. Each neuron has a position, health and bias and a variable number of dendrites. Each dendrite has a position, health and weight. The behaviour of a neuron soma is governed by a single evolved program. In addition each dendrite is governed by another single evolved program. The soma program decides the values of new soma variables position, health and bias based on previous values, the average over all dendrites belonging to the neuron of dendrite health, position and weight and an external input called *problem type*. The latter is a floating point value that indicates the neuron type. The dendrite program updates dendrite health, position and weight based on previous values, the parent neuron's health, position and bias and problem type. When the evolved programs are executed, neurons can change, die replicate and grow more dendrites and their dendrites can also change or die.



than 1.0. The thinking behind the problem type input is that since output neurons are dedicated to a particular computational problem, they should be given information that relates to this, so that the identical neural programs can behave differently according to the computational problem they are associated with.

Bias refers to an input to the neuron activation function which is added to the weighted sum of inputs (i.e. it is unweighted). The soma program updates its own health, bias and position based on these inputs. These are indicated by primed sym-

bols in Fig. 1). The user can decide between three different ways of using the program outputs to update the neural variables. Which is most effective is a research question. The update method is decided by a user defined parameter called Incr_{opt} (see Sec. 3.4) which defines how neuron variables are adjusted by the evolved programs (using user-defined incremental constants or otherwise).

Every dendrite belonging to each neuron is controlled by an evolved dendrite program. The inputs to this program are the health, weight and position of the dendrite and also the health, bias and position of the parent neuron. In addition as mentioned earlier, dendrite programs can also receive the problem type of the parent neuron The the evolved dendrite program decides how the health, weight and position of the dendrite are to be updated.

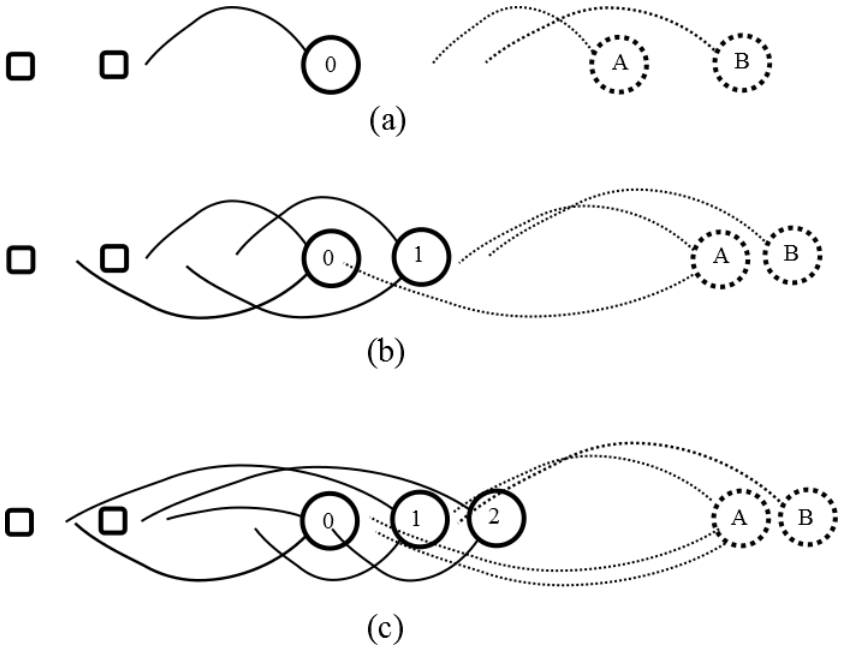
In the model, all the neuron and dendrite parameters (weights, bias, health, position and problem type) are defined by numbers in the range $[-1,-1]$.

A fictitious developmental example is shown in Fig. 2. The initial state of the brain is represented in (a). Initially there is one non-output neuron with a single dendrite. The curved nature of the dendrites is purely for visualisation. In reality the dendrites are horizontal lines emanating from the centre of neurons and of various lengths. When extracting ANNs the dendrites are assumed to connect to their nearest neuron on the left (referred to as 'snapping'). Output neurons are only allowed to connect to non-output neurons or the first input (by default, if their dendrites lie on the left of the leftmost non-output neuron). Thus the ANN that can be extracted from the initial brain, has three neurons. The non-output neuron is connected to the second input and both output neurons are connected via their single dendrite to the non-output neuron.

Fig. 2(b) shows the brain after a single developmental step. In this step, the soma program and dendrite programs are executed in each neuron. The non-output neuron (labeled 0) has replicated to produce non-output neuron (labeled 1) it has also grown a new dendrite. Its dendrites connect to both inputs. The newly created non-output neuron is identical to its parent except that its position is a user-defined amount, MN_{inc}, to the right of the parent and its health is set to 1 (an assumption of the model). Both its dendrites connect to the second input. It is assumed that the soma programs running in the two output neurons A and B have resulted in both output neurons having moved to the right. Their dendrites have also grown in length. Neuron A's first dendrite is now connected to neuron one. In addition, neuron A has high health so that it has grown a new dendrite. Every time a new dendrite grows it is given a weight and health equal to 1.0. Also its new dendrite is given a position equal to half the parent neuron's position. These are assumptions of the model. This its new dendrite is connected to neuron zero.Neuron B's only dendrite is connected to neuron one.

Fig. 2(c) shows the brain after a two developmental steps. The dendrites of neuron zero have changed little and it is still connected in the same way as the previous step. Neuron one's dendrites have both changed. The first one has become longer but remains connected to the first input. The second dendrite has become shorter but it still snaps to the second input. Neuron one has also replicated as a result of its health being above the replication threshold. It gets dendrites identical to its parent,

Fig. 2 Example showing a developing brain. The squares on the left represent the inputs. The solid circles indicate non-output neurons. Non-output neurons have solid dendrites. The dotted circles represent output neurons. Output neuron's dendrites are also dotted. In this example we assume that only output neurons are allowed to move. The neurons, inputs and dendrites are all bound to the interval [-1,1]. Dendrites connect to nearest neurons or inputs on the *left* of their position (*snapping*). (a) shows the initial state of the brain. (b) shows the brain after one developmental step and (c) shows it after two developmental steps.



its position is again incremented to the right of its parent and its health is set to 1.0. Its first dendrite connects to input one and its second dendrite to neuron zero. Output neuron A has gained a dendrite, due to its health being above the dendrite birth threshold. The new dendrite stretches to a position equal to half of its parent neuron. So it connects to neuron zero. The other two dendrites remain the same and they connects to neuron one and zero respectively. Finally, output neuron B's only dendrite has extended a little but still snaps to neuron one. Note, that at this stage neuron two is not connected to this is redundant. It will be stripped out of the ANN that is extracted from the brain.

3.1 Model parameters

The model necessarily has a large number of user-defined parameters these are shown in Table 1.

The total number of neurons allowed in the network is bounded between a user-defined lower (upper)bound NN_{min} (NN_{max}). The number of dendrites each neuron can have is bounded by user-defined lower (upper) bounds denoted by DN_{min} (DN_{max}). These parameters ensure that the number of neurons and connections per neuron remain in well-defined bounds, so that a network can not eliminate itself or grow too large. The initial number of neurons is defined by N_{init} and the initial number of dendrites per neuron is given by ND_{init}.

If the health of a neuron falls below (exceeds) a user-defined threshold, NH_{death} (NH_{birth}) the neuron will be deleted (replicated). Likewise, dendrites are subject to user defined health thresholds, DH_{death} (DH_{birth}) which determine whether the dendrite will be deleted or a new one will be created. Actually, to determine dendrite birth the parent *neuron* health is compared with DH_{birth} rather than dendrite health. This choice was made to prevent the potential very rapid growth of dendrite numbers.

When the soma or dendrite programs are run the outputs are used to decide how to adjust the neural and dendrite variables. The amount of the adjustments are decided by the six user-defined δ parameters.

The number of developmental steps in the two developmental phases ('pre' learning and 'while' learning) are defined by the parameters, NDS_{pre} and NDS_{whi}. The number of learning epochs is defined by N_{ep}. Note that the pre-learning phase of development, 'pre', can have different incremental constants (i.e. δs) to the learning epoch phase, 'while'.

In some cases, neurons will collide with other neurons (by occupying a small interval around an existing neuron) and the neuron has to be moved by a certain increment until no more collisions take place. This increment is given by MN_{inc}.

The places where external inputs are provided is predetermined uniformly within the region between -1 and I_u. The parameter I_u defines the upper bound of their position. Also output neurons are initially uniformly distributed between the parameter O_l and 1. However, depending on a user-defined option the output neurons as with other neurons can move according to the neuron program. All neurons are marked as to whether they provide an external output or not. In the initial network there are N_{init} non-output neurons and N_o output neurons, where N_o denotes the number of outputs required by the computational problem being solved.

Finally, the neural activation function (hyperbolic tangent) and the sigmoid function (which is used in nonlinear incremental adjustment of neural variables) have a slope constant given by α.

Table 1 Table of neural model constants and their meanings.

Symbol	Meaning
$NN_{min}(NN_{max})$	Min. (Max.) allowed number of neurons
N_{init}	Initial number of non-output neurons
$DN_{min}(DN_{max})$	Min. (Max.) number of dendrites per neuron
ND_{init}	Initial number of dendrites per neuron
$NH_{death}(NH_{birth})$	Neuron health thresholds for death (birth)
$DH_{death}(DH_{birth})$	Dendrite health thresholds for death (birth)
δ_{sh}	Soma health increment (pre, while)
δ_{sp}	Soma position increment (pre, while)
δ_{sb}	Soma bias increment (pre, while)
δ_{dh}	Dendrite health increment (pre, while)
δ_{dp}	Dendrite position increment (pre, while)
δ_{dw}	Dendrite weight increment (pre, while)
NDS_{pre}	Number of developmental steps before epoch
NDS_{whi}	Number of 'while' developmental steps during epoch
N_{ep}	Number of learning epochs
MN_{inc}	Move neuron increment if collision
I_u	Max. program input position
O_l	Min. program output position
α	Sigmoid/Hyperbolic tangent exponent constant

3.2 Developing the brain and evaluating the fitness

An overview of the algorithm used for training and developing the ANNs is given in Overview 1.

Overview 1 Overview of fitness algorithm

```
1:  function FITNESS
2:      Initialise brain
3:      Load 'pre' development parameters
4:      Update brain NDS_pre times by running soma and dendrite programs
5:      Load 'while' developmental parameters
6:      repeat
7:          Update brain NDS_whi times by running soma and dendrite programs
8:          Extract ANN for each benchmark problem
9:          Apply training inputs and calculate accuracy for each problem
10:         Fitness is the normalised average accuracy over problems
11:         If fitness reduces terminate learning loop and return previous fitness
12:     until N_ep epochs complete
13:     return fitness
14: end function
```

The brain is always initialised with at least as many neurons as the maximum number of outputs over all computational problems. Note, all problem outputs are represented by a unique neuron dedicated to the particular output. However, the

maximum and initial number of non-output neurons can be chosen by the user. Non-output neurons can grow change or give birth to new dendrites. Output neurons can change but not die or replicate as the number of output neurons is fixed by the choice of computational problems. The detailed algorithm for training and developing the ANN is given in Algorithm 1.

3.3 Updating the brain

Updating the brain is the process of running the soma and dendrite programs once in all neurons and dendrites (i.e. it is a single developmental step). Doing this will cause the brain to change and after all changes have been carried out a new updated brain will be produced. This replaces the previous brain. Overview algorithm 2 gives a high-level overview of the update brain process.

Overview 2 Update brain overview

```
1: function UPDATEBRAIN
2:     Run soma program in non-output neurons to update soma
3:     Ensure neuron does not collide with neuron in updated brain
4:     Run dendrite program in all non-output neurons
5:     If neuron survives add it to updated brain
6:     If neuron replicates ensure new neuron does not collide
7:     Add new neuron to updated brain
8:     Run soma program in output neurons to update soma
9:     Ensure neuron does not collide
10:    Run dendrite program in all output neurons
11:    If neuron survives add it to updated brain
12:    Replace old brain with updated brain
13: end function
```

Sect. 14.1 presents a more detailed version of how the brain is updated at each developmental step (see Algorithm 2) and gives details of the neuron collision avoidance algorithm.

3.4 Running and updating the soma

The UPDATEBRAIN program calls the RUNSOMA program to determine how the soma changes in each developmental step. As we saw in Fig. 1(a) the seven soma program inputs are: neuron health, position and bias, the averaged position, weight and health of the neuron's dendrites and the problem type. Once the evolved CGP soma program is run the soma outputs are returned to the brain update program. These steps are shown in Overview 2.

Overview 2 Running the soma: algorithm overview

1: **function** RUNSOMA
2: Calculate average dendrite health, position and weight
3: Gather soma program inputs
4: Run soma program
5: Return updated soma heath, bias and position
6: **end function**

The detailed version of the RUNSOMA function can be found in Sect. 14.3. The RUNSOMA function uses the soma program outputs to adjust the health, position and bias of the soma according to three user-chosen options defined by a variable Incr_{opt}.This is carried out by the UPDATENEURON overview Alg. 3.

Overview 3 Update neuron algorithm overview

1: **function** UPDATENEURON
2: Assign original neuron variables to parent variables
3: Assign outputs of soma program to health, position and bias
4: Depending on Incr_{opt} get increments
5: If soma program outputs > 0 (≤ 0) then incr(decr.) parent variables
6: Assign parent variables to neuron
7: Bound health, position and bias
8: **end function**

3.5 Updating the dendrites and building the new neuron

This section is concerned with running the evolved dendrite programs. In every dendrite, the inputs to the dendrite program have to be gathered. The dendrite program is executed and the outputs are used to update the dendrite. This is carried out by a function called RUNDENDRITE. Note, in RUNALLDENDRITES we build the completely updated neuron from the updated soma and dendrite variables. The simplified algorithm for doing this is shown in overview algorith 4. The more detailed version is available in Sect. 14.5.

Overview Alg. 4 (in line 9) uses the updated dendrite variables obtained from running the evolved dendrite program to adjust the dendrite variables (according to the incrementation option chosen). This function is shown in the overview Alg. 5. The more detailed version is available in Sect. 14.5.

The RUNDENDRITE function begins by assigning the dendrite's health, position and weight to the parent dendrite variables. It writes the dendrite program outputs to the internal variables health, weight and position. It respectively carries out the increments or decrements of the parent dendrite variables according whether the cor-

Overview 4 An overview of the RUNALLDENDRITES algorithm which runs all dendrite programs and uses all updated variables to build a new neuron.

```
1: function RUNALLDENDRITES
2:     Write updated soma variables to new neuron
3:     if Old soma health > DH_birth then
4:         Generate a dendrite for new neuron
5:     end if
6:     for all Dendrites do
7:         Gather dendrite program inputs
8:         Run dendrite program to get updated dendrite variables
9:         Run dendrite to get updated dendrite
10:        if Updated dendrite is alive then
11:            Add updated dendrite to new neuron
12:            if Maximum number of dendrites reached then
13:                Stop processing any more dendrites
14:            end if
15:        end if
16:    end for
17:    if All dendrites have died then
18:        Give new neuron the first dendrite of the old neuron
19:    end if
20: end function
```

responding dendrite program outputs are greater than or less than or equal to zero. After this it bounds those variables. Finally, it updates the dendrites health, weight and position provided the adjusted health is above the dendrite death threshold.

Overview 5 Change dendrites according to the evolved dendrite program.

```
1: function RUNDENDRITE
2:     Assign original dendrite variables to parent variables
3:     Assign outputs of dendrite program to health, position and weight
4:     Depending on Incr_opt get increments
5:     If dendrite program outputs > 0 (≤0) then incr(decr.) parent variables
6:     Assign parent variables to neuron
7:     Bound health, position and weight
8:     if (health > DH_death) then
9:         Update dendrite variables
10:        Dendrite is alive
11:    else
12:        Dendrite is dead
13:    end if
14:    Return updated dendrite variables and whether dendrite is alive
15: end function
```

We saw in the fitness function that we extract conventional ANNs from the evolved brain. The way this is accomplished is as follows.

Since we share inputs across problems we set the number of inputs to be the maximum number of inputs that occur in the computational problem suite. If any problem has less inputs the extra inputs are set to zero.

The next phase is to go through all dendrites of the neurons to determine which inputs or neurons they connect to. To generate a valid neural network we assume that dendrites are automatically connected to the nearest neuron or input on the left. We refer to this as "snapping". The dendrites of non-output neurons are allowed to connect to either inputs or other non-output neurons on their left. However, output neurons are only allowed to connect to non-output neurons on their left. It is not desirable for the dendrites of output neurons to be connected directly to inputs, however, when output neurons are allowed to move, they may only have inputs on their left. In this case the output neuron dendrite neuron will be connected to the first external input to the ANN network (by default).

The detailed version of the ANN extraction process is given in Sect. 14.6.

4 Cartesian GP

The two neural programs are represented and evolved using a form of Genetic Programming (GP) known as Cartesian Genetic Programming (CGP). CGP [31, 28] is a form of GP in which computational structures are represented as directed, often acyclic graphs indexed by their Cartesian coordinates. Each node may take its inputs from any previous node or program input (although recurrent graphs can also be implemented see [45]). The program outputs are taken from the output of any internal node or program input. In practice, many of the nodes described by the CGP chromosome are not involved in the chain of connections from program input to program output. Thus, they do not contribute to the final operation of the encoded program, these inactive, or "junk", nodes have been shown to greatly aid the evolutionary search [30, 46, 47]. The representational feature of inactive genes in CGP is also closely related to the fact that it does not suffer from bloat [27].

In general, the nodes described by CGP chromosomes are arranged in a rectangular $r \times c$ grid of nodes, where r and c respectively denote the user-defined number of rows and columns. In CGP, nodes in the same column are not allowed to be connected together. CGP also has a connectivity parameter l called "levels-back" which determines whether a node in a particular column can connect to a node in a previous column. For instance if $l = 1$ all nodes in a column can only connect to nodes in the previous column. Note that levels-back only restricts the connectivity of nodes; it does not restrict whether nodes can be connected to program inputs (terminals). However, it is quite common to adopt a linear CGP configuration in which $r = 1$ and $l = c$. This was done in our investigations here. CGP chromosomes can describe multiple input multiple output (MIMO) programs with a range of node functions and arities. For a detailed description of CGP, including its current developments and applications, see [28]. Both the soma and dendrite program have 7 inputs and 3 outputs. (see Fig. 1). The function set chosen for this study are defined

over the real-valued interval [-1.0, 1.0]. Each primitive function takes up to three inputs, denoted z_0, z_1 and z_2. The functions are defined in Table 2.

Table 2 Node function gene values, mnemonic and function definition

Value	mnemonic	Definition
0	abs	$\|z_0\|$
1	sqrt	$\sqrt{\|z_0\|}$
2	sqr	${z_0}^2$
3	cube	${z_0}^3$
4	exp	$(2exp(z_0+1)-e^2-1)/(e^2-1)$
5	sin	$\sin(z_0)$
6	cos	$\cos(z_0)$
7	tanh	$\tanh(z_0)$
7	inv	$-z_0$
9	step	**if** $z_0 < 0.0$ **then** 0 **else** 1.0
10	hyp	$\sqrt{({z_0}^2+{z_1}^2)/2}$
11	add	$(z_0+z_1)/2$
12	sub	$(z_0-z_1)/2$
13	mult	$z_0 z_1$
14	max	**if** $z_0 >= z_1$ **then** z_0 **else** z_1
15	min	**if** $z_0 <= z_1$ **then** z_0 **else** z_1
16	and	**if** $(z_0 > 0.0$ **and** $z_1 > 0.0)$ **then** 1.0 **else** -1.0
17	or	**if** $(z_0 > 0.0$ **or** $z_1 > 0.0)$ **then** 1.0 **else** -1.0
18	rmux	**if** $z_2 > 0.0$ **then** z_0 **else** z_1
19	imult	$-z_0 z_1$
20	xor	**if** $(z_0 > 0.0$ **and** $z_1 > 0.0)$ **then** -1.0 **else if** $(z_0 < 0.0$ **and** $z_1 < 0.0)$ **then** -1.0 **else** 1.0
21	istep	**if** $z_0 < 1.0$ **then** 0 **else** -1.0
22	tand	**if** $(z_0 > 0.0$ **and** $z_1 > 0.0)$ **then** 1.0 **else if** $(z_0 < 0.0$ **and** $z_1 < 0.0)$ **then** -1.0 **else** 0.0
23	tor	**if** $(z_0 > 0.0$ **or** $z_1 > 0.0)$ **then** 1.0 **else if** $(z_0 < 0.0$ **or** $z_1 < 0.0)$ **then** -1.0 **else** 0.0

5 Benchmark problems

In this study, we evolve neural programs that build ANNs for solving three standard classification problems. The problems are cancer, diabetes and glass. The definitions of these problems are available in the well-known UCI repository of machine learning problems[1]. These three problems were chosen because they are well-studied and also have similar numbers of inputs and a small number of classes. Cancer has 9 real attributes and two Boolean classes. Diabetes has 8 real attributes and two Boolean

[1] `https://archive.ics.uci.edu/ml/datasets.html`

classes. Glass has 9 real attributes and six Boolean classes. The specific datsets chosen were cancer1.dt, diabetes1.dt and glass1.dt which are described in the PROBEN suite of problems [2]. Since, for each benchmark problem we extract an ANN the order of presentation of the benchmark problems is unimportant.

6 Experiments and Results

The long-term aim of this research is to explore effective ways to *develop* ANNs. The work presented here is a just a beginning and there are many aspects that need to be investigated in the future (see Sect. 12). The specific research questions we have focused on are:

- Are multiple learning epochs more effective than a single epoch?
- Should neurons be allowed to move?
- Should evolved program outputs update neural variables directly or should they determine user-defined increments in those variables (linear or non-linear)?

To answer these questions a series of experiments were carried out to investigate the impact of various aspects of the neural model on classification accuracy. Twenty evolutionary runs of 20,000 generations of a 1+5-ES were used. Genotype lengths for soma and dendrite programs were chosen to be 800 nodes. Goldman mutation [10, 11] was used which carries out random point mutation until an active gene is changed. For these experiments a subset of allowed node functions were chosen as they appeared to give better performance. These were: step, add, sub, mult, xor, istep. The remaining experimental parameters are shown in Table 3:

Four types of experiments were carried out to investigate the utility of neuron movement. Acronyms describe these experiments. AMA means all neuron movement was allowed (both non-output and output neurons). OMA means only the movement of output neurons is allowed. NOMA means only the movement of non-output neurons is allowed and finally, AMD means all movement of neurons is disallowed. In addition, we examined three ways of incrementing or decrementing neural variables. In the first the outputs of evolved programs determines directly the new values of neural variables (position, health, bias, weight), that is to say there is no incremental adjustment of neural variables. In the second, the variables are incremented or decremented in user-defined amounts (the deltas in Table 1). In the third, the adjustments to the neural variables are nonlinear (they are adjusted using a sigmoid function). It should be noted that the scenario AMD does not imply that all neurons remain in the fixed positions that they were initially given. The collision avoidance mechanism and the birth of new neurons means that neurons will be assigned different positions during development. However, the neuron positions can not be adjusted by incrementing neuron position.

[2] `https://publikationen.bibliothek.kit.edu`

Table 3 Table of neural model parameters.

Parameter	Value
$NN_{min}(NN_{max})$	0 (20)
N_{init}	5
$DN_{min}(DN_{max})$	1 (40)
ND_{init}	5
NDS_{pre}	8
NDS_{whi}	3
NDS_{aft}	0
N_{ep}	1
MN_{inc}	0.03
I_u	-0.6
O_l	0.8
α	1.5
'Pre' development parameters	
$NH_{death}(NH_{birth})$	-0.6 (0.308)
$DH_{death}(DH_{birth})$	-0.404772 (-0.2012)
δ_{sh}	0.1
δ_{sp}	0.1
δ_{sb}	0.07
δ_{dh}	0.1
δ_{dp}	0.2032
δ_{dw}	0.1
'While' development parameters	
$NH_{death}(NH_{birth})$	-0.58 (0.8)
$DH_{death}(DH_{birth})$	-0.38 (0.85)
δ_{sh}	0.01
δ_{sp}	0.01
δ_{sb}	0.0402
δ_{dh}	0.01
δ_{dp}	0.01
δ_{dw}	0.02029

7 Tables of results

The mean, median, maximum and minimum accuracies achieved over 20 evolutionary runs when all neurons are allowed to move are shown in Table 4. We can see that the best values of mean, median, maximum and minimum are all obtained when only output neurons are allowed to move. The mean, median, maximum and minimum are shown for each individual problem (cancer, diabetes and glass) in Table 5.

Table 6 shows how the results for OMA compare with the performance of 179 classifiers (covering 17 families) [8][3]. The figures are given just to show that the results for the developmental ANNs are respectable and are especially encouraging considering that the evolved developmental programs build classifiers for three different classification problems simultaneously.

[3] The paper gives a link to the detailed performance of the 179 classifiers which contain the figures given in the table

Table 4 Training and testing accuracy for various neuron movement scenarios. All neurons allowed to move (AMA), only output neurons are allowed to move (OMA), only non-output neurons are allowed to move (NOMA) and no neurons are allowed to move (AMD).

Acc.	AMA Train (Test)	OMA Train (Test)	NOMA Train (Test)	AMD Train (Test)
Mean	0.7093 (0.6959)	0.7456 (0.7206)	0.6929 (0.6803)	0.6920 (0.6821)
Median	0.7066 (0.7020)	0.7481 (0.7329)	0.6886 (0.6954)	0.6929 (0.6950)
Maximum	0.7598 (0.7539)	0.7854 (0.7740)	0.7617 (0.7643)	0.7363 (0.7245)
Minimum	0.6627 (0.6275)	0.7022 (0.6498)	0.6254 (0.6028)	0.6575 (0.6089)

Table 5 Training and testing accuracy on individual problems when only output neurons are allowed to move.

Acc.	Cancer Train (Test)	Diabetes Train (Test)	Glass Train (Test)
Mean	0.9397 (0.9534)	0.7094 (0.6622)	0.5879 (0.5462)
Median	0.9471 (0.9598)	0.7031 (0.6510)	0.5888 (0.5849)
Maximum	0.9657 (0.9942)	0.7526 (0.7500)	0.6636 (0.6415)
Minimum	0.8771 (0.8391)	0.6693 (0.6094)	0.4766 (0.3774)

Table 6 Comparison of test accuracies on three classification problems. OMA compared with huge suite of classification methods as described in [8]

Acc.	Cancer ML (OMA)	Diabetes ML (OMA)	Glass ML (OMA)
Mean	0.935(0.9534)	0.743(0.6622)	0.610(0.5462)
Maximum	0.974(0.9942)	0.790(0.7500)	0.785(0.6415)
Minimum	0.655(0.8391)	0.582(0.6094)	0.319(0.3774)

8 Comparisons and statistical significance

The results for the four experimental scenarios are presented graphically in Figs. 3 and 4. Maximum outliers are shown as asterisks and minimum outliers as filled squares. The ends of the whiskers are set at 1.5*IQR above the third quartile and at at 1.5*IQR below the first quartile, where IQR is the inter quartile range (Q3-Q1). Clearly, the figures show that allowing only the output neurons to move (OMA) produces the best results both on the training data set and the test data set. Also, in this scenario there is a high level of generalisation as the results on the unseen data set are close to the training results.

The Wilcoxon Ranked-Sum test (WRS) was used to assess the statistical difference between pairs of experiments. In this test, the null hypothesis is that the results (best accuracy) over the multiple runs for the two different experimental conditions are drawn from the same distribution and have the same median. If there is a statistically significant difference between the two then null hypothesis is false with a degree of certainty which depends on the smallness of a calculated statistic called a p-value. However, in the WRS before interpreting the p-value one needs to calculate another statistic called Wilcoxon's W value. This value needs to be compared with calculated values which depend on the number of samples in each experiment.

Fig. 3 Results for four experiments which allow or disallow neurons to move. The four neuron movement scenarios are: all neurons allowed to move (AMA), only output neurons are allowed to move (OMA), only non-output neurons are allowed to move (NOMA) and no neurons are allowed to move (AMD). The figure shows classification accuracy on training set.

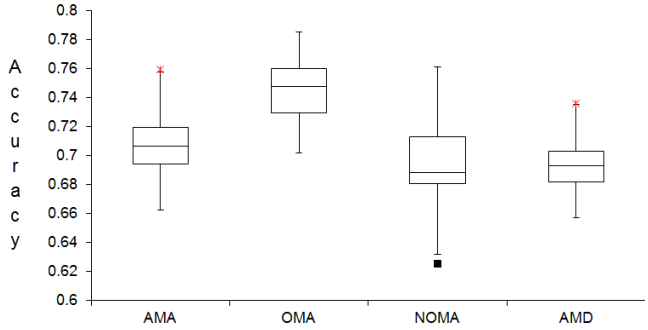
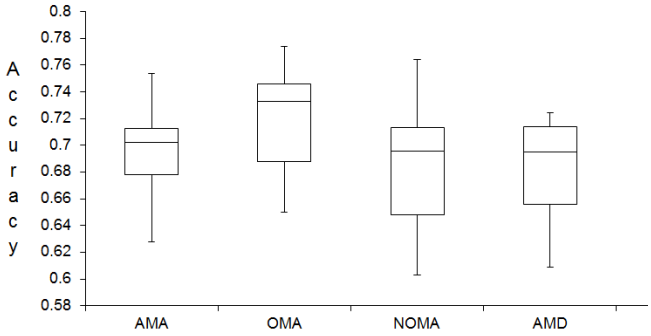


Fig. 4 Results on test set for four experiments which allow or disallow neurons to move.



Results are statistically significant when the calculated W-value is less than or equal to certain critical values for W[48]. The critical values depend on the sample sizes and the p-value. We used a publicly available Excel spreadsheet for doing these calculations[4]. The critical W-values can be calculated in two ways: one-tailed or two-tailed. The two-tailed test is appropriate here as we are interested in whether one experiment is better than another *(*and vice versa).

For example, in Table 7 the calculated W-value is 0 and the critical W-value for for the paired sample sizes of 20 (number of runs) with p-value less than 0.001 is 21 (assuming a two-tailed test)[5]. The p-value gives a measure of the certainty with which the null hypothesis can be accepted. Thus the lower the value the more likely that the two samples come from different distributions (i.e. are statistically different). Thus in this case, the probability that the null hypothesis can be rejected is 0.999.

[4] `http://www.biostathandbook.com/wilcoxonsignedrank.html`

[5] `http://www.real-statistics.com/statistics-tables/wilcoxon-signed-ranks-table/`

The results of these tests are shown in Table 7 and Table 8. Comparing OMA with AMD shows that there is an large advantage to allowing output neurons to move. Indeed, allowing only output neurons to move is statistically significantly better than either only allowing non-outputs neurons to move (NOMA) or allowing all neurons to move (AMA). This is true both for testing and training. However, as expected the differences are even more significant in training than testing. Also using a linear increment is statistically significantly better than using a nonlinear increment. Nonlinear increment is only marginally better than no increment (i.e. $Incr_{opt}$=0).

It is important to understand how the experimental parameters shown in Table 3 were discovered. They were found by carrying out many experiments in the *OMA scenario*. It turned out that when small changes in parameters had a clear improvement on the quality of the first evolutionary run they significantly improved the average performance over all twenty runs. [6]. This was fortuitous in that one could investigate the effect of changing parameters quickly by observing the performance of the first evolutionary run. However, it is possible that the parameters found were particular to the OMA scenario and that a similar process of tuning in the other scenarios would probably improve the results in those scenarios. Ideally, one would tune the parameters in each scenario (OMA, AMA, etc.) and compare results for the best parameter set for each scenario. This would be computationally prohibitive.

Table 7 Statistical comparison of *training* results from experiments (Wilcoxon Rank-Sum two-tailed).

Question	Expt. A	Expt. B	W	W critical	P-value	significant
output movement v. no. movement?	OMA	AMD	0	21	$p < 0.001$	very
output movement v. non-output movement?	OMA	NOMA	1	21	$p < 0.001$	very
output movement v. all movement?	OMA	AMA	33	37	$0.005 < p < 0.01$	yes
Linear v. nonlinear incr.	OMA	OMA-non-lin	19	37	$p < 0.001$	very
Non-linear v. no increment?	OMA-non-lin	OMA no incr.	54	69	$0.05 < p < 0.1$	weakly

9 Evolved developmental programs

The average number of *active* nodes in the soma and dendrite programs for the OMA experiments was respectively, 56.75 and 55.0. Thus the programs are relatively sim-

[6] Indeed, sometimes adjustments to a parameter in the fourth decimal place had a significant effect

Table 8 Statistical comparison of *test* results from experiments (Wilcoxon Rank-Sum two-tailed).

Question	Expt. A	Expt. B	W	W critical critical	P-value	significant
output movement v. no movement?	OMA	AMD	35	37	$0.005 < p < 0.01$	yes
output movement v. non-output movement?	OMA	NOMA	44	52	$0.02 < p < 0.05$	yes
output movement v. all movement?	OMA	AMA	53	60	$0.05 < p < 0.1$	yes

ple. It is also possible that the graphs can be logically reduced to even simpler forms. The graphs of the active nodes in the CGP graphs for the best evolutionary run in the OMA scenario are shown in Figs. 5 – 6. The red input connections between nodes indicate the first input in the subtraction operation. This is the only node operation where node input order is important.

Fig. 5 Best evolved soma program when only output neurons can move. The input nodes are: soma heath (sh), soma bias (sb), soma position (sp), average dendrite health (adh), average dendrite weight (adw), average dendrite position (adp) and problem type (pt). The output nodes are: soma health (SH), soma bias (SB) and soma position (SP).

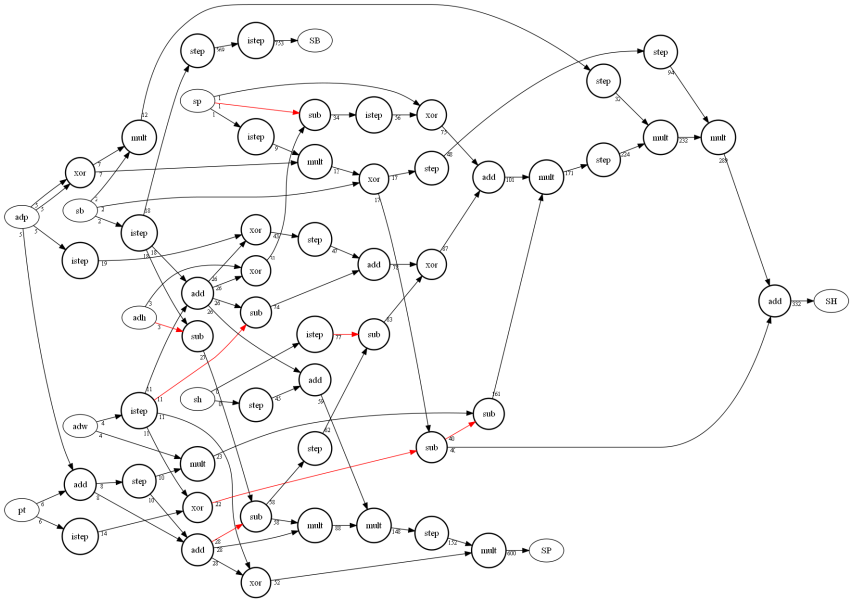
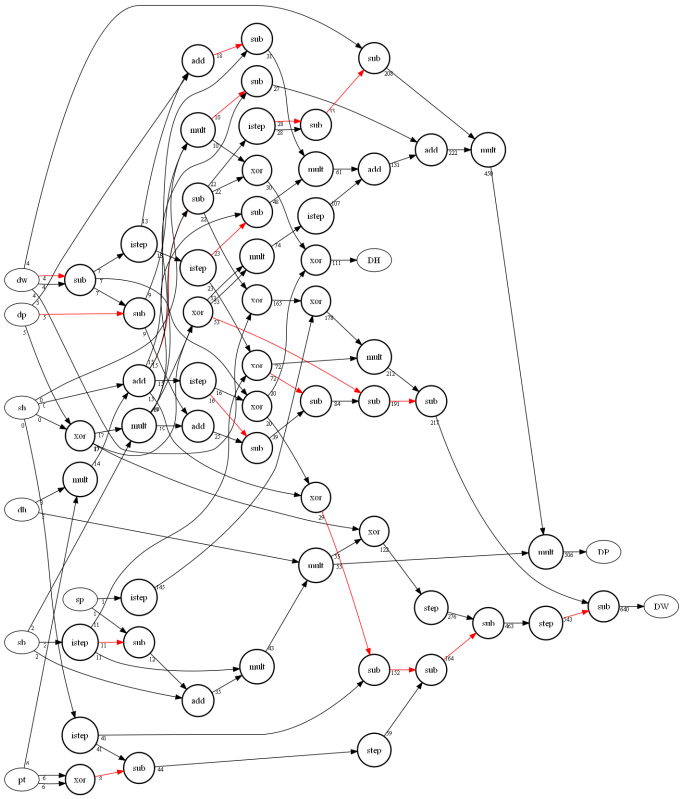


Fig. 6 Best evolved dendrite program when only output neurons can move. The input nodes are: soma heath (sh), soma bias (sb), soma position (sp), dendrite health (dh), dendrite weight (dw), dendrite position (dp) and problem type (pt). The output nodes are: dendrite health (DH), dendrite weight (DW) and dendrite position (DP).



10 Developed ANNs for each classification problem

The ANNs for the evolutionary run with only output neuron movement allowed were extracted (using Alg. 9) and can be seen in Figs. 7 – 9. The plots ignore connections with weight equal to zero. The average training accuracy of these three networks is 0.78538 (best) and the average test accuracy is 0.7459. In the figures, each neuron is labeled with the problems it belongs to (cancer, diabetes, glass) and it is also labeled with the neuron ID (in blue). Using these labels makes it easy to identify which neurons are shared between problems. Note that output neurons are not allowed to be shared.

The ANN which classifies the cancer data has 6 neurons with IDs: 9, 10, 12, 13, 19, 20. Neurons 9 and 13 are shared with the glass ANN, Neurons 10 and 12 are

shared over all three problems. Class 0 of the cancer dataset is provided by a simple function. The first attribute in the dataset is multiplied by a single weight and then is the only input to a biased tanh neuron.

The ANN classifier for the diabetes dataset has 5 neurons with IDs: 10, 12, 15, 21, 27. Neurons with IDs 10 and 12 are shared across all problems. Neuron 15 is shared with glass ANN.

The ANN classifier for the glass dataset has 15 neurons with IDs: 9, 10, 11, 12, 13, 14, 15, 16, 18, 22, 23, 24, 25, 26, 28. There are 4 neurons that are shared.

An interesting feature is that pairs of neurons often have multiple connections. This is equivalent to a single connection where the weighted value is the sum of the individual connections weights. This phenomenon was also observed in CGP encoded and evolved ANNs [44].

11 Evolving neural learning programs

The fitness function (see overview algorithm 1) included the possibility of learning epochs. In this section we present and discuss results when a number of learning epochs have been chosen. The task for evolution is then to construct two neural programs that develop ANNs that improve with each learning epoch. The aim is to find a general learning algorithm in which the ANNs change and improve with each learning epoch beyond the limited number of epochs used in training. The experimental parameters required to investigate were changed from those used previously when there were no learning epochs. For the experiments here we retained most of the previous parameters but altered the number of learning epochs and the delta parameters in the 'while' loop. The new parameters are shown in Table 9.

Table 9 Table of changed neural model parameters when using multiple learning epochs.

Parameter	Value
N_{ep}	10
δ_{sh}	0.0
δ_{sp}	0.0
δ_{sb}	0.0
δ_{dh}	0.0
δ_{dp}	0.00106
δ_{dw}	0.0

Informal experiments were undertaken to investigate suitable neural parameters when using multiple learning epochs. The best results appeared to be obtained when only the dendrite length was incrementally adjusted. As before, results are quite sensitive to exact values for these parameters. It is interesting and surprising to observe that adjustment of weights only in while phase does not produce as good results as adjusting dendrite length. Twenty evolutionary runs were carried out using these parameters and the results are shown in Table 10.

Fig. 7 Developed ANN for cancer dataset. This datset has 9 attributes and two outputs. The numbers inside the circles are the neuron bias. Above the bias the problems which share the neuron are shown. Below the bias the neuron ID (in blue) is shown. If any attributes are not present it means they are unused. The training accuracy is 0.9571 and the test accuracy is 0.9828.

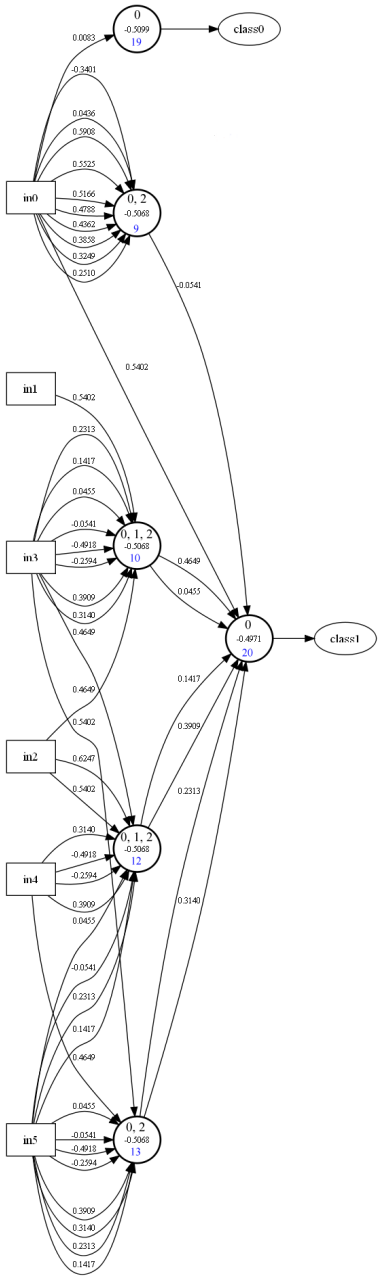
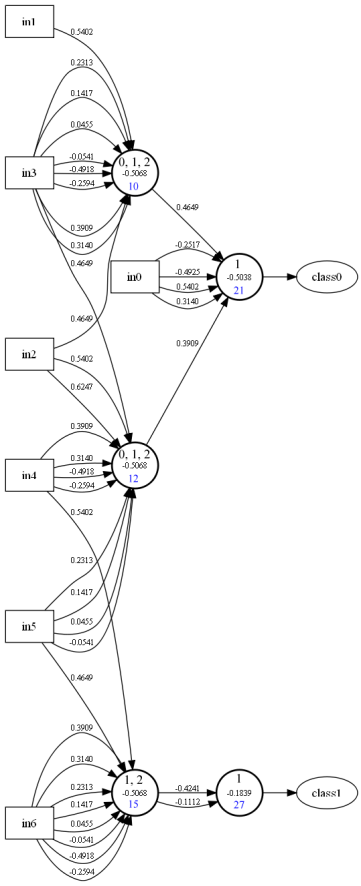


Fig. 8 Developed ANN for diabetes dataset.This dataset has 8 attributes and two outputs. The numbers inside the circles are the neuron bias. Above the bias the problems which share the neuron are shown. Below the bias the neuron ID (in blue) is shown. Attributes not present are unused. The training accuracy is 0.7448 and the test accuracy is 0.6510



In Table 10 we compare the results with multiple learning epochs with no learning epochs. Using no learning epochs gives better results. However, the results with multiple learning epochs is reasonable despite the fact that the task is much more difficult, effectively one is to trying evolve a *learning algorithm*. It is possible that further experimentation with developmental parameters could produce better results with multiple epochs.

In Figure 10 we examine how the accuracy of the classifications varies with learning epochs. We set the maximum number of epochs to 30 now to see if learning continues beyond the upper limit used during evolution (10). We can see that both the test and training classification accuracy increases with each epoch up to 10 epochs

Fig. 9 Developed ANN for glass dataset. This dataset has 9 attributes and six outputs. The numbers inside the circles are the neuron bias. Above the bias the problems which share the neuron are shown. Below the bias the neuron ID (in blue) is shown. Attributes not present are unused. The training accuracy is 0.6542 and the test accuracy is 0.6038

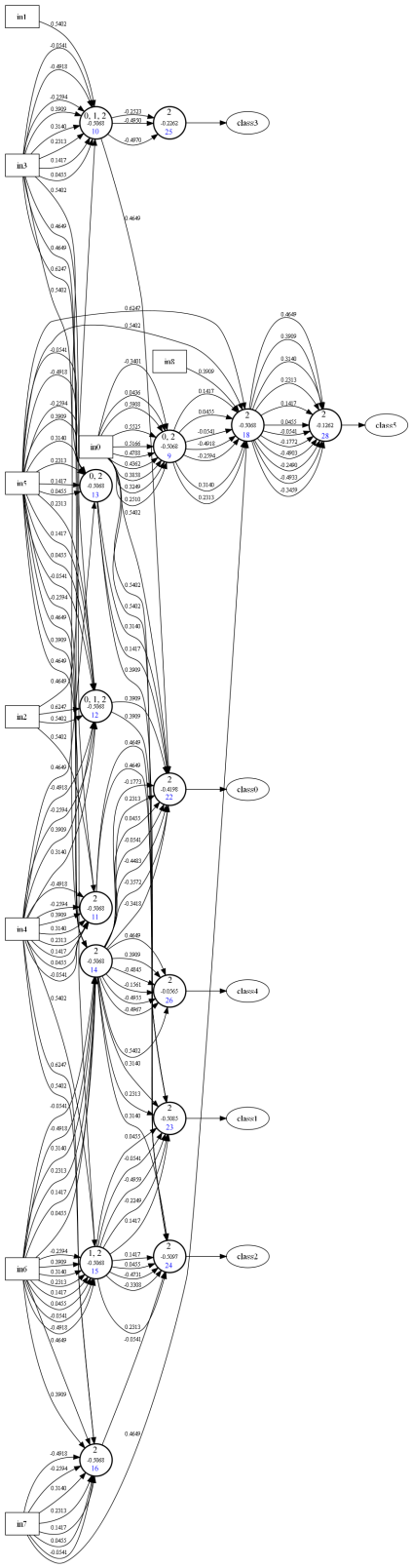
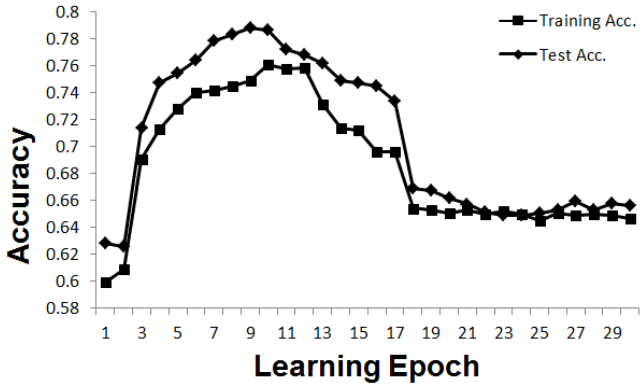


Table 10 Training and testing accuracy for ten learning epochs versus no learning epochs when only output neurons are allowed to move.

Acc.	Learning epochs Train (Test)	No learning epochs Train (Test)
Mean	0.7226 (0.6482)	0.7456 (0.7206)
Median	0.7298 (0.6767)	0.7481 (0.7329)
Maximum	0.7609 (0.7864)	0.7854 (0.7740)
Minimum	0.6613 (0.3919)	0.7022 (0.6498)

and there is a gradual decline in accuracy after this point. However, at 18 epochs the accuracy stabilises to an accuracy of 0.65. Interestingly, the test accuracy is always better than the training accuracy. We obtained several evolved solutions in which training accuracy increased at each epoch until the imposed maximum number of epochs, however, as yet none of these were able to improve beyond the limit.

Fig. 10 Variation of classification accuracy for training and testing with learning epoch when only output movement is allowed.



12 Further work

It remains unclear why better results can not at present be obtained when evolving developmental programs with multiple epochs. Neither is it clear why programs can be evolved that continuously improve the developed ANNs over a number of epochs (i.e. 10) yet do not improve subsequently. It is worth contrasting the model discussed in this chapter with previous work on Self-Modifying CGP [14]. In SMCGP phenotypes can be iterated to produce a sequence of programs or phenotypes. The fitness was *accumulated* over all the correct test cases summed over all the iterations. In the problems studied (i.e. even-n parity, producing π) there was also a notion of

perfection. For instance in the parity case perfection meant that at each iteration it produced the next parity case (with more inputs) perfectly. If at the next iteration, the appropriate parity function was not produced, then the iteration stopped. In the work discussed here, the fitness is not cumulative. At each epoch, the fitness is the average accuracy of the classifiers over the three classification problems. If the fitness reduces at the next epoch, then the epoch loop is terminated. However, in principle, we could sum the accuracies at each epoch and if an accuracy at a particular epoch is reduced, terminate the epoch loop. Summing the accuracies would give reward to developmental programs that produced the best *history* of developmental changes.

At present, the developmental programs do not receive a reward signal during multiple epochs. This means that the task for evolution is to continuously improve developed ANNs without being supplied with a reward signal. However, one would expect that as the fitness increases at each epoch the number of changes that need to be made to the developed ANNs should decrease. This suggests that supplying the fitness at the previous epoch to the developmental programs might be useful. In fact this option has already been implemented but as yet evidence is inconclusive that this produces improved results.

While learning over multiple epochs, we have assumed that the developmental parameters should be fixed (i.e. they are chosen before the development loop - see line 5 of Overview algorithm 1). However, it is not clear that this should be so. One could argue that during early learning topological changes in the brain network are more important and weight changes more important in later phases of learning. This suggests that at each step of the learning loop one could load developmental parameters, this would allow control of each epoch of learning.

The neural variables that are given as inputs to the CGP developmental programs are an assumption of the model. For the soma these are: health, bias, position, problem type and average dendrite health, position and weight. For the dendrite they are: dendrite health, weight, position, problem type and soma health, bias and position. Further experimental work needs to be undertaken to determine whether they all are useful. The program written already has the inclusion of any of these variables as an option.

There are also many assumptions made in quite small aspects of the whole model. When new neurons or dendrites are born what should the initial values of the neural variables be? What are the best upper bounds on the number of neurons and dendrites? Currently, dendrite replication is decided by comparing the parent *neuron* health with DH_{birth} rather than comparing dendrite health with this threshold. If dendrite health was compared with a threshold it could rapidly lead to very large numbers of dendrites. Many choices have been made that need to be investigated in more detail.

There are also very many parameters in the model and experiment has shown that results can be very sensitive to these. Thus further experimentation is required to identify good choices for these parameters.

A fundamental issue is how to handle inputs. In the classification problems the number of inputs is given by the problem with the most attributes, problems with less are given the value zero for those inputs. This could be awkward if the prob-

lems have hugely varying numbers of inputs. Is there another way of handling this? Perhaps one could borrow more ideas from SMCGP and make all input connections access inputs using pointer to a circular register of inputs. Every time a neuron connected to an input, a global pointer to the register of inputs would be incremented.

So far, we have examined the utility of the developmental model on three classification problems. However, the aim of the work is to produce general problem solving on many different kinds of computational problems. Clearly, a favourable direction to go is to expand the list of problems and problem types. How much neuron sharing would take place across problems of different types (e.g. classification and real-time control)? Would different kinds of problems cause whole new sub-networks to grow?

Currently the neurons exist in a one-dimensional space however it would be relatively straightforward to extend it to two or even three spatial dimensions. This remains for future work.

Eventually, the aim is to create developmental networks of spiking neurons. This would allow a study of activity dependent development [33] which is an extremely important aspect of biological brains.

13 Conclusions

We have presented a conceptually simple model of a developmental neuron in which neural networks develop over time. Conventional ANNs can be extracted from these networks. We have shown that an evolved pair of programs can produce networks that can solve multiple classification problems reasonably well. Multiple-problem solving is a new domain for investigating more general developmental neural models.

14 Appendix: Detailed algorithms

14.1 Developing the brain and evaluating the fitness

The detailed algorithm for developing the brain and assessing its fitness is shown in Alg. 1 There are two stages to development. The first (which we refer to as 'pre') occurs prior to a learning epoch loop (lines 3-6). While the second phase (referred to as 'while') occurs inside a learning epoch loop (lines 9-12).

Lines 13-22 are concerned with calculating fitness. For each computational problem an ANN is extracted from the underlying brain. This is carried by a function $ExtractANN(problem, OutputAddress)$ which is detailed in Alg. 9. This function extracts a feedforward ANN corresponding to each computional problem (this is stored in a phenotype which we do not detail here). The array $OutputAddress$ stores

the addresses of the output neurons associated with the computational problem. It is used together with the phenotype to extract the network of neurons that are required for the computational problem. Then the input data is supplied and the outputs of the ANN calculated. The class of a data instance is determined by the largest output. The learning loop (lines 8-29) develops the brain and exits if the fitness value (in this case classification accuracy) reduces (lines 23-27 in Alg. 1). One can think of the 'pre' development phase as growing a neural network prior to training. The 'while' phase is a period of development within the learning phase. N_{ep} denotes the user-defined number of learning epochs. N_p represents the number of problems in the suite of problems being solved. $N_{ex}(p)$ denotes the number of examples for each problem. A is the accuracy of prediction for a single training instance. F is the fitness over all examples. TF is the accumulated fitness over all problems. Fitness is normalised (lines 20 and 22).

Algorithm 1 Develop network and evaluate fitness

```
 1: function FITNESS
 2:     Initialise brain
 3:     Use 'pre' parameters
 4:     for s = 0 to s < NDS_pre do                  # develop prior to learning
 5:         UpdateBrain
 6:     end for
 7:     TF_prev = 0
 8:     for e = 0 to e < N_ep do                     # learning loop
 9:         Use 'while' parameters                   # learning phase
10:         for s = 0 to s < NDS_whi do
11:             UpdateBrain
12:         end for
13:         TF = 0                                   # initialise total fit
14:         for p = 0 to p < N_p do
15:             ExtractANN(p, OutputAddress)         # Get ANN for problem p
16:             F = 0                                # initialise fit
17:             for t = 0 to t < N_ex(p) do
18:                 F = F + Acc                      # sum acc. over instances
19:             end for
20:             TF = TF + F/N_ex(p)                  # sum normalised acc. over problems
21:         end for
22:         TF = TF/N_p                              # normalise total fitness
23:         if TF < TF_prev then                     # has fitness reduced?
24:             TF = TF_prev                         # return previous fitness
25:             Break                                # terminate learning loop
26:         else
27:             TF_prev = TF                         # update previous fitness
28:         end if
29:     end for
30:     return TF
31: end function
```

14.2 Developing the brain and evaluating the fitness

Algorithm 2 shows the update brain process. This algorithm is run at each developmental step. It runs the soma and dendrite programs for each neuron and from the previously existing brain creates a new version (*NewBrain*) which eventually overwrites the previous brain at the last step (lines 52-53).

Algorithm 2 Update brain

```
1: function UPDATEBRAIN
2:     NewNumNeurons = 0
3:     for i = 0 to i < NumNeurons do                # get number and addresses of neurons
4:         if (Brain[i].out = 0) then
5:             NonOutputNeuronAddress[NumNonOutputNeurons] = i
6:             increment NumNonOutputNeurons
7:         else
8:             OutputNeuronAddress[NumOutputNeurons] = i
9:             increment NumOutputNeurons
10:        end if
11:    end for
12:    for i = 0 to i < NumNonOutputNeurons do       # process non-output neurons
13:        NeuronAddress = NonOutputNeuronAddress[i]
14:        Neuron = Brain[NeuronAddress]
15:        UpdatedNeurVars = RunSoma(Neuron)         # get new position, health and bias
16:        if (DisallowNonOutputsToMove) then
17:            UpdatedNeurVars.x = Neuron.x
18:        else
19:            UpdatedNeurVars.x = IfCollision(NewNumNeurons,NewBrain,UpdatedNeurVars.x)
20:        end if
21:        UpdatedNeuron = RunAllDendrites(Neuron, UpdatedNeurVars)
22:        if (UpdatedNeuron.health > NH_death) then   # if neuron survives
23:            NewBrain[NewNumNeurons] = UpdatedNeuron
24:            Increment NewNumNeurons
25:            if (NewNumNeurons = NN_max-NumOutputNeurons) then
26:                Break                             # exit non-output neuron loop
27:            end if
28:        end if
29:        if (UpdatedNeuron.health > NH_health) then  # neuron replicates
30:            UpdatedNeuron.x = UpdatedNeuron.x+MN_inc
31:            UpdatedNeuron.x = IfCollision(NewNumNeurons, NewBrain, UpdatedNeuron.x)
32:            NewBrain[NewNumNeurons] = CreateNewNeuron(UpdatedNeuron)
33:            Increment NewNumNeurons
34:            if (NewNumNeurons = NN_max - NumOutputNeurons) then
35:                Break                             # exit non-output neuron loop
36:            end if
37:        end if
38:    end for
39:    for i = 0 to i < NumOutputNeurons do          # process output neurons
40:        NeuronAddress = OutputNeuronAddress[i]
41:        Neuron = Brain[NeuronAddress]
42:        UpdatedNeurVars = RunSoma(Neuron)         # get new position, health and bias
43:        if (DisallowOutputsToMove) then
44:            UpdatedNeurVars.x = Neuron.x
45:        else
46:            UpdatedNeurVars.x = IfCollision(NewNumNeurons,NewBrain,UpdatedNeurVars.x)
47:        end if
48:        UpdatedNeuron = RunAllDendrites(UpdatedNeuron)
49:        NewBrain[NewNumNeurons] = UpdatedNeuron
50:        Increment NewNumNeurons
51:    end for
52:    NumNeurons = NewNumNeurons
53:    Brain = NewBrain
54: end function
```

Alg. 2 starts by analyzing the brain to determine the addresses and numbers of non-output and output neurons (lines 3-11). Then the non-output neurons are processed. The evolved soma program is executed and it returns a neuron with updated values for the neuron position, health and bias. These are stored in the variable $UpdatedNeurVars$.

If the user-defined option to disallow non-output neuron movement is chosen then the updated neuron position is reset to that before the soma program is run (lines 16-17). Next the evolved dendrite programs are executed in all dendrites. The algorithmic details are given in Alg. 6 (See Sect. 3.5).

The neuron health is compared with the user-defined neuron death threshold NH_{death} and if the health exceeds the threshold the neuron survives (see lines 22-28). At this stage it is possible that the neuron has been given a position that is identical to one of the neurons in the developing brain (*NewBrain*) so one needs a mechanism for preventing this. This is accomplished by Alg. 3 (Lines 19 and 46). It checks whether a collision has occurred and if so an increment MN_{inc} is added to the position and then it is bound to the interval [-1, 1]. In line 23 the updated neuron is written into *NewBrain*. A check is made in line 25 to see if the allowed number of neurons has been reached, if so the non output neuron update loop (lines 12 to 38) is exited and the output neuron section starts (lines 39 to 51). If the limit on numbers of neurons has not been reached, the updated neuron may replicate depending on whether its health is above the user-defined threshold, NH_{health} (line 29). The position of the new born neuron is immediately incremented by MN_{inc} so that it does not collide with its parent (line 30). However, its position needs to be checked also to see if it collides with any other neuron, in which case its position is incremented again until a position is found that causes no collision. This is done in the function IFCOLLISION.

In CREATENEWNEURON (see line 32) the bias, the incremented position and dendrites of the parent neuron are copied into the child neuron. However, the new neuron is given a health of 1.0 (the maximum value). The algorithm examines the non-output neurons (lines 39-51) and again is terminated if the allowed number of neurons is exceeded. The steps are similar to those carried out with non-output neurons, except that output neurons can not either die or replicate as their number is fixed by the number of outputs required by the computational problem being solved.

The details of the neuron collision avoidance mechanism is shown in Alg. 3.

14.3 Running the soma

The UPDATEBRAIN program calls the RUNSOMA program (Alg. 4) to determine how the soma changes in each developmental step. The seven soma program inputs comprising the neuron health, position and bias, the averaged position, weight and health of the neuron's dendrites and the problem type are supplied to the CGP encoded soma program (line 12). The array *ProblemTypeInputs* stores

Algorithm 3 Move neuron if it collides with another.

```
1:  function IFCOLLISION(NumNeurons, Brain, NeuronPosition)
2:      NewPosition = NeuronPosition
3:      collision = 1
4:      while collision do
5:          collision = 0
6:          for i = 0 to j < NumNeurons do
7:              if (| NeuronPosition - Brain[i].x | < 1.e-8) then
8:                  collision = 1
9:              end if
10:             if collision then
11:                 break
12:             end if
13:         end for
14:         if collision then
15:             NewPosition = NewPosition+MN_inc
16:         end if
17:     end while
18:     if collision then
19:         NewPosition = Bound(NewPosition)
20:     end if
21:     return NewPosition
22: end function
```

NumProblems+1 constants equally spaced between -1 and 1. These are used to allow output neurons to know what computational problem they belong to.

The soma program has three outputs relating to the position, health and bias of the neuron. These are used to update the neuron (line 13).

Algorithm 4 RunSoma(Neuron)

```
1:  function RUNSOMA(Neuron)
2:      AvDendritePosition = GetAvDendritePosition(Neuron)
3:      AvDendriteWeight = GetAvDendriteWeight(Neuron)
4:      AvDendriteHealth = GetAvDendriteHealth(Neuron)
5:      SomaProgramInputs[0] = Neuron.health
6:      SomaProgramInputs[1] = Neuron.x
7:      SomaProgramInputs[2] = Neuron.bias
8:      SomaProgramInputs[3] = AvDendritePosition
9:      SomaProgramInputs[4] = AvDendriteWeight
10:     SomaProgramInputs[5] = AvDendriteHealth
11:     SomaProgramInputs[6] = ProblemTypeInputs[WhichProblem]
12:     SomaProgramOutputs = SomaProgram(SomaProgramInputs)
13:     UpdatedNeuron = UpdateNeuron(Neuron, SomaProgramOutputs)
14:     return UpdatedNeuron.x, UpdatedNeuron.health, UpdatedNeuron.bias
15: end function
```

14.4 Changing the Neuron Variables

The UPDATENEURON algorithm (5) updates the neuron properties of health, position and bias according to three user-chosen options defined by a variable Incr_{opt}. If this is zero, then the soma program outputs determine directly the updated values of the soma's health, position and bias. If Incr_{opt} is one or two, the updated values of the soma are changed from the parent neuron's values in an incremental way. This is either a linear or nonlinear increment or decrement depending on whether the soma program's outputs are greater than or less than or equal to zero (lines 8 to 16). The magnitudes of the increments is defined by the user-defined constants: δ_{sh}, δ_{sp}, δ_{sb} and sigmoid slope parameter, α (see Table 1).

The increment methods described in Algorithm 5 change neural variables, so action needs to be taken to force the variables to strictly lie in the interval $[-1,1]$. We call this 'bounding' (lines 34-36).This is accomplished using a hyperbolic tangent function.

14.5 Running all dendrite programs and building a new neuron

Alg. 6 takes an existing neuron and creates a new neuron using the updated soma variables, position, health and bias which are stored in $UpdateNeurVars$ (from Alg. 4) and the updated dendrites which result from running the dendrite program in all the dendrites. Initially (line 3-5), the updated soma variables are written into the updated neuron. The number of dendrites in the updated neuron is set to zero. In lines 8-11, the health of the non-updated *neuron* is examined and if it is above the *dendrite health threshold* for birth, a new dendrite is generated and the updated neuron gains a dendrite. If so, the neuron gains a dendrite created by a function $GenerateDendrite()$. This assigns a weight, health and position to the new dendrite. The weight and health is set to one and the position set to half the parent neuron position. These choices appeared to give good results.

Lines 12-33 are concerned with processing the dendrite program in all the dendrites of the non-updated neuron and updating the dendrites. If the updated dendrite has a health above its death threshold then it survives and gets written into the updated neuron (lines 22-28). Updated dendrites do not get written into the updated neuron if it already has the maximum allowed number of dendrites (line 25-27). In lines 30-33 a check is made as to whether the updated neuron has no dendrites. If this is so, it is given one of the dendrites of the non-updated neuron. Finally, the updated neuron is returned to the calling function.

Alg. 6 calls the function RUNDENDRITE (line 21). This function is detailed in Alg. 7. It changes the dendrites of a neuron according to the evolved dendrite program. It begins by assigning the dendrites health, position and weight to the parent dendrite variables. It writes the dendrite program outputs to the internal variables health, weight and position. Then in lines 8-16 it defines the possible increments in health, weight and position that will be used to increment or decrement the parent

Algorithm 5 Neuron update function

```
 1: function UPDATENEURON(Neuron, SomaProgramOutputs)
 2:     ParentHealth = Neuron.health
 3:     ParentPosition = Neuron.x
 4:     ParentBias = Neuron.bias
 5:     health = SomaProgramOutputs[0]
 6:     position = SomaProgramOutputs[1]
 7:     bias = SomaProgramOutputs[2]
 8:     if (Incr_opt = 1) then                          # calculate increment
 9:         HealthIncrement = δ_sh
10:         PositionIncrement = δ_sp
11:         BiasIncrement = δ_sb
12:     else if (Incr_opt = 2) then
13:         HealthIncrement = δ_sh*sigmoid(health, α)
14:         PositionIncrement = δ_sp*sigmoid(position, α)
15:         BiasIncrement = δ_sb*sigmoid(bias, α)
16:     end if
17:     if (Incr_opt > 0) then                          # apply increment
18:         if (health > 0.0) then
19:             health = ParentHealth + HealthIncrement
20:         else
21:             health = ParentHealth - HealthIncrement
22:         end if
23:         if (position > 0.0) then
24:             position = ParentPosition + PositionIncrement
25:         else
26:             health = ParentPosition - PositionIncrement
27:         end if
28:         if (bias > 0.0) then
29:             bias = ParentBias + BiasIncrement
30:         else
31:             bias = ParentBias - BiasIncrement
32:         end if
33:     end if
34:     health = Bound(health)
35:     position = Bound(position)
36:     bias = Bound(bias)
37:     return health, position and bias
38: end function
```

variables according to the user defined incremental options (linear or non-linear).In lines 17-33 it respectively carries out the increments or decrements of the parent dendrite variables according whether the corresponding dendrite program outputs are greater than or less than or equal to zero. After this it bounds those variables. Finally, in lines 37-44 it updates the dendrites health, weight and position provided the adjusted health is above the dendrite death threshold (in other words it survives). Note that if $Incr_{opt} = 0$ then there is no incremental adjustment and the health, weight and position of the dendrites are just bounded (lines 34-36).

Algorithm 6 Run the evolved dendrite program in all dendrites

```
1: function RUNALLDENDRITES(Neuron, DendriteProgram, NewSomaPosition, NewSoma-
   Health, NewSomaBias)
2:     WhichProblem = Neuron.isout
3:     OutNeuron.x = NewSomaPosition
4:     OutNeuron.health = NewSomaHealth
5:     OutNeuron.bias = NewSomaBias
6:     OutNeuron.isout = WhichProblem
7:     OutNeuron.NumDendrites = 0
8:     if (Neuron.health > DH_birth ) then
9:         OutNeuron.dendrites[NumDendrites] = GenerateDendrite()
10:        Increment OutNeuron.NumDendrites
11:    end if
12:    for i = 0 to i < OutNeuron.NumDendrites do
13:        DendriteProgramInputs[0] = Neuron.health
14:        DendriteProgramInputs[1] = Neuron.x
15:        DendriteProgramInputs[2] = Neuron.bias
16:        DendriteProgramInputs[3] = Neuron.dendrites[i].health
17:        DendriteProgramInputs[4] = Neuron.dendrites[i].weight
18:        DendriteProgramInputs[5] = Neuron.dendrites[i].position
19:        DendriteProgramInputs[6] = ProblemTypeInputs[WhichProblem]
20:        DendriteProgramOutputs = DendriteProgram(DendriteProgramInputs)
21:        UpdatedDendrite = RunDendrite(Neuron, DendriteProgramOutputs)
22:        if (UpdatedDendrite.isAlive) then
23:            OutNeuron.dendrites[NumDendrites] = UpdatedDendrite
24:            increment OutNeuron.NumDendrites
25:            if (OutNeuron.NumDendrites > MaxNumDendrites) then
26:                break
27:            end if
28:        end if
29:    end for
30:    if (OutNeuron.NumDendrites = 0) then          # if all dendrites die
31:        OutNeuron.dendrites[0] = Neuron.dendrites[0]
32:        OutNeuron.NumDendrites = 1
33:    end if
34:    return OutNeuron
35: end function
```

Alg. 2 uses a function CREATENEWNEURON to create a new neuron if the neuron health is above a threshold. This function is described in Alg. 8. It makes the new born neuron the same as the parent (note, its position will be adjusted by the collision avoidance algorithm) except that it is given a health of one. Experiments suggested that this gave better results.

Algorithm 7 Change dendrites according to the evolved dendrite program

```
 1: function RUNDENDRITE(Neuron, WhichDendrite, DendriteProgramOutputs)
 2:     ParentHealth = Neuron.dendrites[WhichDendrite].health
 3:     ParentPosition = Neuron.dendrites[WhichDendrite].x
 4:     ParentWeight = Neuron.dendrites[WhichDendrite].weight
 5:     health = DendriteProgramOutputs[0]
 6:     weight = DendriteProgramOutputs[1]
 7:     position = DendriteProgramOutputs[2]
 8:     if (Incr_opt = 1) then
 9:         HealthIncrement = δ_dh
10:         WeightIncrement = δ_dw
11:         PositionIncrement = δ_dp
12:     else if (Incr_opt = 2) then
13:         HealthIncrement = δ_dh*sigmoid(health, α)
14:         WeightIncrement = δ_dw*sigmoid(weight, α)
15:         PositionIncrement = δ_dp*sigmoid(position, α)
16:     end if
17:     if (Incr_opt > 0) then
18:         if (health > 0.0) then
19:             health = ParentHealth + HealthIncrement
20:         else
21:             health = ParentHealth - HealthIncrement
22:         end if
23:         if (position > 0.0) then
24:             position = ParentPosition + PositionIncrement
25:         else
26:             health = ParentPosition - PositionIncrement
27:         end if
28:         if (weight > 0.0) then
29:             weight = ParentWeight + BiasIncrement
30:         else
31:             weight = ParentWeight - BiasIncrement
32:         end if
33:     end if
34:     health = Bound(health)
35:     position = Bound(position)
36:     weight = Bound(weight)
37:     if (health > DH_death) then
38:         UpdatedDendrite.weight = weight
39:         UpdatedDendrite.health = health
40:         UpdatedDendrite.x = position
41:         UpdatedDendriteisAlive = 1
42:     else
43:         UpdatedDendriteisAlive = 0
44:     end if
45:     return UpdatedDendrite and UpdatedDendriteisAlive
46: end function
```

14.6 Extracting conventional ANNs from the evolved brain

In algorithm 1, a conventional feed-forward ANN is extracted from the underlying collection of neurons (line 15). The algorithm for doing this is shown in algorithm 9.

Algorithm 8 Create new neuron from parent neuron

```
 1: function CREATENEWNEURON(ParentNeuron)
 2:     ChildNeuron.NumDendrites = ParentNeuron.NumDendrites
 3:     ChildNeuron.isout = 0
 4:     ChildNeuron.health = 1
 5:     ChildNeuron.bias = ParentNeuron.bias
 6:     ChildNeuron.x = ParentNeuron.x
 7:     for i = 0 to i < ChildNeuron.NumDendrites do
 8:         ChildNeuron.dendrites[i] = ParentNeuron.dendrites[i]
 9:     end for
10: end function
```

Firstly, this algorithm determines the number of inputs to the ANN (line 5). Since inputs are shared across problems the number of inputs is set to be the maximum number of inputs that occur in the computational problem suite. If an individual problem has less inputs than this maximum, the extra inputs are set to 0.0. The brain array is sorted by position. The algorithm then examines all neurons (line 7) and calculates the number of non-output neurons and output neurons and stores the neuron data in arrays $NonOutputNeurons$ and $OutputNeurons$. It also calculates their addresses in the brain array.

The next phase is to go through all dendrites of the non-output neurons to determine which inputs or neurons they connect to (lines 19 to 33). The evolved neuron programs generate dendrites with end positions anywhere in the interval [-1, 1]. The end positions are converted to lengths (line 25). In this step the dendrite position is linearly mapped into the interval [0, 1]. To generate a valid neural network we assume that dendrites are automatically connected to the nearest neuron or input *on the left*. We refer to this as "snapping" (lines 28 and 44). The dendrites of non-output neurons are allowed to connect to either inputs or other non-output neurons on their left. However, output neurons are only allowed to connect to non-output neurons on their left. Algorithm 10 returns the address of the neuron or input that the dendrite snaps to. The dendrites of output neurons are not allowed to connect directly to inputs (see Line 4 of the GETCLOSEST function), however, when neurons are allowed to move, there can occur a situation where an output neuron is positioned so that it is the first neuron on the right of the outputs. In that situation it can only connect to inputs. If this situation occurs then the initialisation of the variable $AddressOfClosest$ to zero in the GETCLOSEST function (line 2) means that all the dendrites of the output neuron will be connected to the first external input to the ANN network. Thus a valid network will still be extracted albeit with a rather useless output neuron. It is expected that evolution will avoid using programs that allow this to happen.

Algorithm 9 stores the information required to extract the ANN in an array called $Phenotype$. It contains the connection addresses of all neurons and their weights (lines 29-30 and 45-46). Finally it stores the addresses of the output neurons ($OutputAddress$) corresponding to the computational problem whose ANN is being extracted (lines 49-52). These define the outputs of the extracted ANNs when

they are supplied with inputs (i.e. in the fitness function when the Accuracy is assessed (see Alg. 1). The *Phenotype* is stored in the same format as Cartesian Genetic Programming (see section 4) and decoded in a similar way to genotypes.

References

1. Astor, J.C., Adami, C.: A development model for the evolution of artificial neural networks. Artificial Life **6**, 189–218 (2000)
2. Balaam, A.: Developmental neural networks for agents. In: Advances in Artificial Life, Proceedings of the 7th European Conference on Artificial Life (ECAL 2003), pp. 154–163. Springer (2003)
3. Boers, E.J.W., Kuiper, H.: Biological metaphors and the design of modular neural networks. Master's thesis, Dept. of Comp. Sci. and Dept. of Exp. and Theor. Psych., Leiden University (1992)
4. Cangelosi, A., Nolfi, S., Parisi, D.: Cell division and migration in a 'genotype' for neural networks. Network-Computation in Neural Systems **5**, 497–515 (1994)
5. Downing, K.L.: Supplementing evolutionary developmental systems with abstract models of neurogenesis. In: Proc. Conf. on Genetic and evolutionary Comp., pp. 990–996 (2007)
6. Eggenberger, P.: Creation of neural networks based on developmental and evolutionary principles. In: W. Gerstner, A. Germond, M. Hasler, J.D. Nicoud (eds.) Artificial Neural Networks — ICANN'97, pp. 337–342 (1997)
7. Federici, D.: A regenerating spiking neural network. Neural Networks **18**(5-6), 746–754 (2005)
8. Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D.: Do we need hundreds of classifiers to solve real world classification problems? J. Mach. Learn. Res. **15**(1), 3133–3181 (2014)
9. French, R.M.: Catastrophic Forgetting in Connectionist Networks: Causes, Consequences and Solutions. Trends in Cognitive Sciences **3**(4), 128–135 (1999)
10. Goldman, B.W., Punch, W.F.: Reducing wasted evaluations in cartesian genetic programming. In: Genetic Programming: 16th European Conference, EuroGP 2013, Vienna, Austria, April 3-5, 2013. Proceedings, pp. 61–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
11. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programmings evolutionary mechanisms. Evolutionary Computation, IEEE Transactions on **19**, 359 – 373 (2015)
12. Gruau, F.: Automatic definition of modular neural networks. Adaptive Behaviour **3**, 151–183 (1994)
13. Gruau, F., Whitley, D., Pyeatt, L.: A comparison between cellular encoding and direct encoding for genetic neural networks. In: Proc. Conf. on Genetic Programming, pp. 81–89 (1996)
14. Harding, S., Miller, J.F., Banzhaf, W.: Developments in cartesian genetic programming: Self-modifying cgp. Genetic Programming and Evolvable Machines **11**(3-4), 397–439 (2010)
15. Hornby, G., Lipson, H., Pollack, J.B.: Generative representations for the automated design of modular physical robots. IEEE Trans. on Robotics and Automation **19**, 703–719 (2003)
16. Hornby, G.S., Pollack, J.B.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life **8(3)** (2002)
17. Huizinga, J., Clune, J., Mouret, J.B.: Evolving neural networks that are both modular and regular: HyperNEAT plus the connection cost technique. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 697–704 (2014)
18. Jakobi, N.: Harnessing Morphogenesis, COGS Research Paper 423. Tech. rep., University of Sussex (1995)
19. Khan, G.M.: Evolution of Artificial Neural Development - In Search of Learning Genes, *Studies in Computational Intelligence*, vol. 725. Springer (2018)

20. Khan, G.M., Miller, J.F.: In search of intelligence: evolving a developmental neuron capable of learning. Connect. Sci. **26**(4), 297–333 (2014)
21. Khan, G.M., Miller, J.F., Halliday, D.M.: Evolution of Cartesian Genetic Programs for Development of Learning Neural Architecture. Evol. Computation **19**(3), 469–523 (2011)
22. Kitano, H.: Designing neural networks using genetic algorithms with graph generation system. Complex Systems **4**, 461–476 (1990)
23. Kodjabachian, J., Meyer, J.A.: Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. IEEE Transactions on Neural Networks **9**, 796–812 (1998)
24. Kumar, S., Bentley, P. (eds.): On Growth, Form and Computers. Academic Press (2003)
25. McCloskey, M., Cohen, N.: Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. The Psychology of Learning and Motivation **24**, 109–165 (1989)
26. McCulloch, Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics **5**, 115–133 (1943)
27. Miller, J.F.: What bloat? cartesian genetic programming on boolean problems. In: Proc. Conf. Genetic and Evolutionary Computation, Late breaking papers, pp. 295–302 (2001)
28. Miller, J.F. (ed.): Cartesian Genetic Programming. Springer (2011)
29. Miller, J.F., Khan, G.M.: Where is the Brain inside the Brain? Memetic Computing **3**(3), 217–228 (2011)
30. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in Cartesian Genetic Programming. IEEE Trans. on Evolutionary Computation **10**(2), 167–174 (2006)
31. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proc. European Conf. on Genetic Programming, *LNCS*, vol. 10802, pp. 121–132 (2000)
32. Miller, J.F., Thomson, P.: A Developmental Method for Growing Graphs and Circuits. In: Proc. Int. Conf. on Evolvable Systems, *LNCS*, vol. 2606, pp. 93–104 (2003)
33. Ooyen, A.V. (ed.): Modeling Neural Development. MIT Press (2003)
34. Ratcliff, R.: Connectionist Models of Recognition and Memory: Constraints Imposed by Learning and Forgetting Functions. Psychological Review **97**, 205–308 (1990)
35. Risi, S., Lehman, J., Stanley, K.O.: Evolving the placement and density of neurons in the HyperNEAT substrate. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 563–570 (2010)
36. Risi, S., Stanley, K.O.: Indirectly encoding neural plasticity as a pattern of local rules. In: From Animals to Animats 11: Conf. on Simulation of Adaptive Behavior (2010)
37. Risi, S., Stanley, K.O.: Enhancing ES-HyperNEAT to evolve more complex regular neural networks. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 1539–1546 (2011)
38. Rust, A., Adams, R., Bolouri, H.: Evolutionary neural topiary: Growing and sculpting artificial neurons to order. In: Proc. Conf. on the Simulation and synthesis of Living Systems, pp. 146–150 (2000)
39. Stanley, K., Miikkulainen, R.: Efficient evolution of neural network topologies. In: Proc. Congress on Evolutionary Computation, vol. 2, pp. 1757–1762 (2002)
40. Stanley, K.O.: Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines **8**, 131–162 (2007)
41. Stanley, K.O., D'Ambrosio, D.B., Gauci, J.: A hypercube-based encoding for evolving large-scale neural networks. Artificial Life **15**, 185–212 (2009)
42. Stanley, K.O., Miikkulainen, R.: A taxonomy for artificial embryogeny. Artificial Life **9(2)**, 93–130 (2003)
43. Suchorzewski, M., Clune, J.: A novel generative encoding for evolving modular, regular and scalable networks. In: Proc. Conf. on Genetic and Evolutionary Computation, pp. 1523–1530 (2011)
44. Turner, A.J., Miller, J.F.: Cartesian Genetic Programming encoded artificial neural networks: A comparison using three benchmarks. In: Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO), pp. 1005–1012 (2013)
45. Turner, A.J., Miller, J.F.: Recurrent cartesian genetic programming. In: Proc. Parallel Problem Solving from Nature, pp. 476–486 (2014)

46. Vassilev, V.K., Miller, J.F.: The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Proc. Int. Conf. on Evolvable Systems, *LNCS*, vol. 1801, pp. 252–263. Springer Verlag (2000)
47. Yu, T., Miller, J.F.: Neutrality and the Evolvability of Boolean function landscape. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 2038, pp. 204–217 (2001)
48. Zar, J.H.: Biostatistical Analysis, 2nd edn. Prentice Hall (1984)

Algorithm 9 The extraction of neural networks from the underlying brain.

```
1: function EXTRACTANN(problem, OutputAddress)
2:     NumNonOutputNeurons = 0
3:     NumOutputNeurons = 0
4:     OutputCount=0
5:     N_i = max(N_i, p)
6:     sort(Brain, 0, NumNeurons-1)                # sort neurons by position
7:     for i = 0 to i < NumNeurons do
8:         Address = i + N_i
9:         if (Brain[i].isout > 0) then            # non-output neuron
10:            NonOutputNeur[NumNonOutputNeur] = Brain[i]
11:            NonOutputNeuronAddress[NumNonOutputNeur]= Address
12:            Increment NumNonOutputNeur
13:        else                                    # output neuron
14:            OutputNeurons[NumOutputNeurons]= Brain[i]
15:            OutputNeuronAddress[NumOutputNeurons]= Address
16:            Increment NumOutputNeurons
17:        end if
18:    end for
19:    for i = 0 to i < NumNonOutputNeur do        # do non-output neurons
20:        Phenotype[i].isout = 0
21:        Phenotype[i].bias = NonOutputNeur[i].bias
22:        Phenotype[i].address = NonOutputNeuronAddress[i]
23:        NeuronPosition = NonOutputNeur[i].x
24:        for j = 0 to j < NonOutputNeur[i].NumDendrites do
25:            Convert DendritePosition to DendriteLength
26:            DendPos = NeuronPosition - DendriteLength
27:            DendPos = Bound(DendPos)
28:            AddressClosest = GetClosest(NumNonOutputNeur, NonOutputNeur, 0, DendPos)
29:            Phenotype[i].ConnectionAddresses[j] = AddressClosest
30:            Phenotype[i].weights[j] = NonOutputNeur[i].weight[j]
31:        end for
32:        Phenotype[i].NumConnectionAddress = NonOutputNeur[i].NumDendrites
33:    end for
34:    for i = 0 to i < NumOutputNeurons do        # do output neurons
35:        i1 = i+NumOutputNeurons
36:        Phenotype[i1].isout = OutputNeurons[i].isout
37:        Phenotype[i1].bias = OutputNeurons[i].bias
38:        Phenotype[i1].address = OutputNeuronAddress[i]
39:        NeuronPosition = OutputNeurons[i].x
40:        for j = 0 to j < OutputNeurons[i].NumDendrites do
41:            Convert DendritePosition to DendriteLength
42:            DendPos = NeuronPosition - DendriteLength
43:            DendPos = Bound(DendPos)
44:            AddressClosest = GetClosest(NumNonOutputNeur, NonOutputNeur, 1, DendPos)
45:            Phenotype[i1].ConnectionAddresses[j] = AddressClosest
46:            Phenotype[i1].weights[j] = OutputNeuron[i].weight[j]
47:        end for
48:        Phenotype[i1].NumConnectionAddress = OutputNeurons[i].NumDendrites
49:        if (OutputNeurons[i].isout == problem+1) then
50:            OutputAddress[OutputCount] = OutputNeuronAddress[i]
51:            Increment OutputCount
52:        end if
53:    end for
54: end function
```

Algorithm 10 Find which input or neuron a dendrite is closest to

```
1: function GETCLOSEST(NumNonOutNeur, NonOutNeur, IsOut, DendPos)
2:     AddressOfClosest = 0
3:     min = 3.0
4:     if (IsOut = 0) then                          # only non-out neurons connect to inputs
5:         for (i = 0 to i < MaxNumInputs) do
6:             distance = DendPos - InputLocations[i]
7:             if distance > 0 then
8:                 if (distance < min) then
9:                     min = distance
10:                    AddressOfClosest = i
11:                end if
12:            end if
13:        end for
14:    end if
15:    for j = 0 to j <NumNonOutputNeur do
16:        distance = DendPos - NonOutNeur[j].x
17:        if distance > 0 then                     # feed-forward connections
18:            if (distance < min) then
19:                min = distance
20:                AddressOfClosest = j + MaxNumInputs
21:            end if
22:        end if
23:    end for
24:    return AddressOfClosest
25: end function
```