Multiple problem solving with evolved developmental neural networks: activity dependence

Julian Francis Miller
University of York, Heslington, York, YO10 5DD, UK
julian.miller@york.ac.uk

Abstract

We construct artificial neural networks (ANNs) by executing evolved programs inside two neural components: the body (soma) and the dendrite. The programs decide whether neurons and their dendrites move, change, die or replicate. When the programs are executed they build a neural structure from which *multiple* conventional ANNs can be extracted each of which can solve a different computational problem. In biological brain development electrical activity strongly influences developmental changes. This is known as *activity dependence*. In this paper, we describe various ways activity dependence could be implemented and investigated. We show that allowing the soma bias to be activity dependent improves performance.

Introduction

Although ANNs were originally inspired by the brain most papers involving neural networks do not use evolution and especially not development. Most ANN models obey the "synaptic dogma" in which learned knowledge is encoded solely in the form of connection strengths (i.e. weights). This gives rise to "catastrophic forgetting" where an ANN loses it ability to solve an earlier problem when it is retrained on a new one. Neuroscience suggests that the synaptic dogma is questionable, as memory in brains is only *weakly* related to synaptic strengths. In brains, synapses are not fixed structures but are constantly pruned away and replaced by new synapses during learning (Smythies, 2002). In addition, structural aspects of the brains are strongly influenced by the electrical activity within the brain. Imaging of living brains reveals continuous structural changes through the forming or breaking of synapses, motile spines, and rerouting of axonal branches in the developing and adult brain. The role of some forms of structural changes appear to contribute to the *homeostasis* of network activity (Butz et al., 2009).

We present a model of a simplified computational equivalent of the biological neuron. Two programs are evolved which *when executed* allow a neural network to grow as many neurons and connections as it needs, adjust its own weights and biases and solve multiple problems. One program (soma) decides whether neurons move, change, die or replicate. The other decides whether dendrites extend, change, die, or replicate. Since developmental programs build networks that change over time we have to define new problem classes that are suitable to evaluate such approaches. We show that the evolved programs can build a network from which *multiple* conventional ANNs can be extracted each of which can solve a different computational problem.

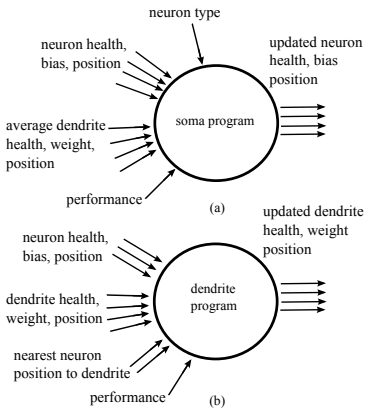
The particular focus of this paper is on forms of *activity dependence* (AD) and whether it improves the performance of the model. AD can be introduced for all the neural and dendrite variables so there are many possibilities to investigate. For instance, dendrite weights can be adjusted in a Hebbian-like manner in which the weight of connections between neurons are increased when both the dendrite signal incident to a neuron and the neuron's output are both high. The weight is decreased if one signal is high and the other low. We investigated AD bias in this paper.

Neuron model

The neural programs are represented using Cartesian Genetic Programming (CGP) (Miller, 2011) in which the program nodes represent mathematical operations, operating on and returning real-values between -1 and 1. Each primitive function takes up to two inputs, denoted z_0, z_1. The functions are as follows. *Step*: if $z_0 < 0$ then 0 else 1. *Add*: $(z_0 + z_1)/2$. *Sub*: $(z_0 - z_1)/2$. *Mult*: $z_0 z_1$. *Xor*: if the sign of both inputs is the same then the output is -1 else 1. *Istep*: if $z0$ is negative, output is 1 else output is 0. These functions were found to be effective in previous work. The programs read variables associated with neurons and dendrites and produce outputs which are used to update those variables. The inputs and outputs to the evolved programs are illustrated in Fig. 1. We refer to the collection of neurons and dendrites as the *brain*. Later we will explain how multiple conventional ANNs can be extracted from the brain and assessed for their effectiveness. Neurons and dendrite variables can only take values between -1.0 and 1.0. Every output required by each computational problem has a dedicated *output neuron*. The other neurons are internal and are

not used to provide outputs from the brain. We refer to these as *non-output neurons*.

Figure 1: The model of a developmental neuron. Each neuron has a two-dimensional position, health, bias and a variable number of dendrites. Each dendrite has a two dimensional position (the growing tip), a health and a weight.



The soma program reads ten variables. Four are neuron variables: x and y position, health and bias. Bias refers to an input to the neuron activation function which is added to the weighted sum of inputs. The soma program is also supplied with averages of properties of its dendritic tree: x and y position, weight and health. The soma program can also read a variable called *neuron type*. For non-output neurons, the neuron type is -1.0. Output neurons are given a value 1.0. This potentially allows a neuron to behave differently depending on whether it is an output neuron or not. Finally, the soma can read the performance score (i.e. fitness) at the previous learning epoch (see next section). The soma program has four outputs: health updater, bias updater, and x and y position updater. The evolved soma program reads its ten inputs and outputs the four soma output update variables. These decide how the corresponding soma variables will be updated. The way this is done is as follows. If any soma updater variable is greater (less) than zero, the corresponding soma variable is incremented (decremented) by the user-defined amounts, δ_{sh}, δ_{sb}, δ_{sp}. There are both 'pre' deltas and 'while' deltas (see next section). After updating, the corresponding variable is squashed into the interval [-1, 1] using a hyperbolic tangent function. In the case of soma health, there is a further step. If it falls below the user-defined death threshold, θ_{nd}, then the neuron will die and not be present in the updated brain. Alternatively, if it happens to be above the user-defined neuron birth threshold, θ_{nb}, then the parent neuron will replicate and an additional neuron will appear in the brain (near to the parent). Thus, the soma evolved programs can change the health, bias or position of the soma and whether the neuron will die, or replicate.

The dendrite program also has ten inputs and is executed inside every dendrite. Three inputs are the parent neuron's bias and x and y positions. Four are the dendrite variables: x and y position, weight and health. The dendrite program is also allowed to read the x and y position of the nearest neuron to the dendrite position. Like the soma, the dendrite can also read the performance score of the brain at the previous epoch (see next section). There are four outputs: health updater, weight updater, and x and y position updater. The evolved dendrite program reads its ten inputs, and outputs the four dendrite output update variables. These decide how the actual dendrite corresponding variables will be updated. If any dendrite updater variable is greater (less) than zero, the corresponding dendrite variable is incremented (decremented) by the user-defined amounts, δ_{dh}, δ_{dw}, δ_{dp}. After updating the corresponding variable is squashed using a hyperbolic tangent function. There are also user-defined thresholds for dendrite birth and death, θ_{db}, θ_{dd}.

Developing the brain and evaluating the fitness

The algorithm used for training and developing the ANNs is given in Alg. 1. 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. Output neurons can change, but not die or replicate as the number of output neurons is fixed by the choice of computational problems. The number of developmental steps are defined by the parameters, NDS_{pre} and NDS_{whi}. The 'pre' learning phase is an initial phase of development where the brain is not tested in any way (lines 4-6). While in the 'while' phase the brain is assessed and provides feedback to the developmental process (lines 10-12).

Lines 9 - 30 form the *epoch learning* loop. This loop repeats the entire training developmental process (the 'while loop') for a number of epochs, N_{ep}. Learning epochs allow us to direct evolution to produce a pair of programs that cause the developing ANN to learn. The neural programs can read the performance of the brain at the previous learning epoch. The learning loop only continues while the training accuracy does not decrease (lines 25-29). If it does, the algorithm stops and returns the training score of the previous epoch.

Note that at each epoch, a performance value is determined corresponding to each individual benchmark problem and is an input to the soma and dendrite programs for *output* neurons. If a neuron is not an output neuron then the average fitness at the previous epoch is given as an input to the soma and dendrite programs. The performance signal is intended to act as a reward to the developmental process, triggering changes in the brain when necessary. We extract conventional ANNs from the developed brain in the following way (line 15 in Alg. 1). First only the *relevant* inputs to the target problem in hand are given their randomly pre-assigned positions. Thus different problems can not connect to inputs

Algorithm 1 Fitness algorithm.

```
1:  function FITNESS
2:      Initialise brain
3:      Load 'pre' development parameters
4:      PrevFitness = 0
5:      for NDS_pre times do
6:          Run soma/dendrite programs to update brain
7:      end for
8:      Load 'while' developmental parameters
9:      for epoch = 1 to N_ep do
10:         for NDS_whi times do
11:             Run evolved programs to update brain
12:         end for
13:         TotFitness = 0
14:         for p = 1 to NumBenchmarkProblems do
15:             Extract ANN
16:             Fitness(p) = 0
17:             for NT(p) training cases do
18:                 Make activity dependent changes
19:                 Fitness(p) = Fitness(p) + FitInstance
20:             end for
21:             Fitness(p) = Fitness(p)/NT(p)
22:             TotFitness = TotFitness + Fitness(p)
23:         end for
24:         TotFitness = TotFitness/NumBenchmarkProblems
25:         if TotFitness < PrevFitness then
26:             Break
27:         else
28:             PrevFitness = TotFitness
29:         end if
30:     end for
31:     Return TotFitness
32: end function
```

Algorithm 2 Signal propagation and activity dependence.

```
1:  for neuron i do
2:      W_sum = 0
3:      for dendrite ij do
4:          W_sum = W_sum + W_ij × DS_ij
5:          if Dendrite acitivity dependence then
6:              D =| DS_ij | −θ_v
7:              Brain_ij(v) = Brain_ij(v) − D × δ^act_dv
8:              Bound Brain_ij(v)
9:          end if
10:     end for
11:     W_sum = W_sum + B_i
12:     NS_i = tanh(αW_sum)
13:     if Hebbian learning then
14:         if NS_i > θ_Hebb then
15:             N_high = 1
16:         end if
17:         if NS_i < −θ_Hebb then
18:             N_low = 1
19:         end if
20:         for dendrite ij do
21:             if DS_ij > θ_Hebb then
22:                 D_high = 1
23:             end if
24:             if DS_ij < −θ_Hebb then
25:                 D_low = 1
26:             end if
27:             Both_high = N_high AND D_high
28:             Both_low = N_low AND D_low
29:             D =| N_i − DS_ij |
30:             if Both_high OR Both_low then
31:                 BrainW_ij = BrainW_ij + D × δ^Hebb_inc
32:                 Bound BrainW_ij
33:             else
34:                 BrainW_ij = BrainW_ij × δ^Hebb_mult
35:             end if
36:         end for
37:     end if
38:     if Neuron activity dependence then
39:         D =| NS_i | −θ_v
40:         Brain_i(v) = Brain_i(v) − D × δ^act_sv
41:         Bound Brain_i(v)
42:     end if
43: end for
```

of another target problem.

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*. Since, the dendrite position can be on the right of the parent neuron, before extracting ANNs it is reflected back from the parent position. 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. Although, it is not desirable for the dendrites of output neurons to be connected directly to inputs, when output neurons move, they may only have inputs on their left. $NT(p)$ is the number of training cases for each computational problem, p. Thus we can see what the brain connects to is problem dependent so the same neuron can appear in one ANN and again (with same bias, position, dendrite numbers and weights) in another ANN with possibly different connections (i.e. to other inputs). The extracted ANNs use the hyperbolic tangent activation function. In line 17 of Alg. 1 the signal related changes (activity dependence) in the brain are accumulated. Note, the algorithm calculates what the output of each neuron is (NS_i) in the extracted ANNs but it also makes activity dependent changes to the brain which will affect extracted ANNs on subsequent training instances. There are various ways of making these changes (see Alg.2). W_{ij} is the weight of dendrite j of neuron i *in the extracted ANN*. $Brain_{ij}(v)$ is variable v of dendrite j of neuron i (i.e. v can be health, weight, or position). $BrainW_{ij}$ is the weight of dendrite j of neuron i in the brain. DS_{ij} denotes the signal passing through dendrite j belonging to neuron i. In lines 6-7 the difference between the absolute value of the dendrite signal and a user defined threshold (θ_v) is calculated. Either the variables, weight, health or position of the dendrite in the brain is then *reduced* in magnitude using the user-defined increment corresponding to the chosen

variable, δ_{dv}^{act}). This is a homeostatic mechanism (Butz et al., 2009) where with large signals, variables reduce in size to maintain homeostasis. For brevity we have only shown one activity dependent adjustment. The user can choose to adjust any or all of the three dendrite variables in this way. The weighted sum of signals over all dendrites belonging to a neuron is accumulated and a bias B_i is added (line 11). The neuron signal, NS_i, is then calculated using a user-defined slope parameter, α (line 12). Then if AD is chosen for a neuron (lines 38-42) brain adjustments take place according to whether the magnitude of neuron signal, $\mid NS_i \mid$ exceeds a user-defined theshold, θ_v. In this paper, we examine AD neuron bias. The neuron health or position could also be adjusted in a similar manner. Lines 13-37 are concerned with Hebbian-like learning. Here, if the magnitude of the signal passing along the dendrite and the output of the parent neuron both exceed a threshold (i.e. they agree), then the weight of the dendrite is increased using the user-defined increment, δ_{inc}^{Hebb}. However, if only one exceeds a threshold (i.e. they disagree) then the weight is decreased (using the user-defined multiplier, δ_{mult}^{Hebb}). It should be noted that the model has many parameters many of which are thresholds and allowed increments on neural variables. The frequency of AD changes can be controlled via the corresponding threshold, θ_v. For details, see (Miller et al., 2019).

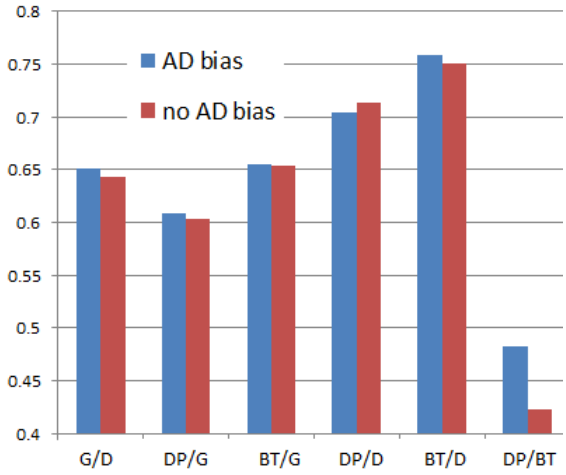
Experiments and Results

Here, we attempt to simultaneously solve pairs of benchmark problems chosen from two classification problems diabetes (D) and glass (G) [1] and two reinforcement learning problems (double-pole balancing and ball throwing). The ball throwing (BT) task is to design a controller which throws a ball as far as possible. There are two inputs, the arm angle from vertical and the angular velocity of the arm. It has two outputs, the applied torque to the arm and an output which decides when to release the ball. The system is simulated for a maximum of 10,000 time steps. The maximum distance the ball can be thrown is 10.202m. BT is considered solved when the thrown distance is greater than or equal to 9.5m (fitness = 0.9312). In double pole balancing (DP), the task is to balance two poles on a moveable cart on a limited track by applying a horizontal force to the cart. The six inputs to the controller are the position and velocity of the cart and the angle and angular velocity of the pole(s). The single output is the force applied to the cart. The system is simulated for a maximum of 100,000 time steps. It is solved if both poles are balanced to within certain limits for this number of steps.

The genotype length is 600 nodes. Goldman mutation was used which carries out random point mutation until an active gene is changed. Multiple problems were solved incrementally the first 5000 generations for problem 1 only, then 5000 for problems 1 and 2. A maximum of 30 neurons were allowed and a maximum of 60 dendrites per neuron. Epoch learning was chosen with $N_{ep} = 8$, $NDS_{pre} = 6$ and $NDS_{whi} = 1$. The increment deltas were as follows: δ_{dh}^{pre} and δ_{nh}^{pre} were 0.2, $\delta_{dw}^{pre} = 0.05$, the remaining deltas ('pre' or 'while') were 0.1. However, when AD bias was allowed, δ_{sb}^{whi} was set to zero. The user defined developmental thresholds were as follows: θ_{db}^{pre} and $\theta_{nb}^{pre} = 0.2$, $\theta_{dd}^{pre} = -0.7$, $\theta_{nd}^{pre} = -0.6$, $\theta_{db}^{whi} = -0.65$, $\theta_{dd}^{whi} = -0.7$, $\theta_{nb}^{whi} = 0.2$ and $\theta_{nd}^{whi} = -0.4$. The signal threshold for neuron bias adjustment, $\theta_{bias} = 0.5$ and the AD soma bias adjustment δ_{nb}^{act} was 0.01

We assessed the utility of activity dependent bias on pairs of benchmark problems (Fig. 2), the results in the two cases were not statistically significantly different. However, it is noteworthy that AD gave superior performance in all but one of the paired experiments (DP/D) and even in that the *median* for AD results was higher.

Figure 2: Comparison of mean fitness for AD bias with no AD bias for six pairs of benchmarks.



Conclusion and Future Work

We have discussed ways that AD can be introduced into the developmental neural model. We found that implementing a form of AD soma bias gives improved performance. AD can be introduced for other neural variables and further work is needed to determine which are most effective. At present we *accumulate* activity related changes in the brain at each epoch, but we could allow the changes to have immediate effect in the ANN during training. In future, it might be better to create new suites of simpler problems for developmental methods, where one can start with simple problems and gradually increase the task complexity.

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

References

Butz, M., Wörgötter, F., and van Ooyen, A. (2009). Activity-dependent structural plasticity. *Brain Research Reviews*, 60(2):287 – 305.

Miller, J. F., editor (2011). *Cartesian Genetic Programming*. Springer.

Miller, J. F., Wilson, D. G., and Cussat-Blanc, S. (2019). Evolving programs to build artificial neural networks. In *From Astrophysics to Unconventional Computation*, pages 23–71. Springer International Publishing.

Smythies, J. (2002). *The Dynamic Neuron*. MIT Press.