Evolving an artificial brain to solve multiple problems

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

Abstract

A developmental method for growing a "brain" like neural network is described. Two programs representing the soma and dendrites of a neuron are represented and evolved using Cartesian Genetic Programming. The programs allow neurons to build a neural network from which multiple conventional artificial neural networks (ANNs) can be extracted. We show that the approach can perform reasonably well on *multiple* simultaneous reinforcement learning problems.

Introduction

ANNs were originally inspired by the brain, however, very few models use evolution and development, both of which are fundamental to the construction of the brain (Miller and Khan, 2011). We describe a simple neural model in which two neural programs construct neural networks. One program represents the neuron soma and the other the dendrite. The soma program decides whether neurons move, change, die or replicate. The dendrite program decides whether dendrites extend, change, die, or 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 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 computational problem. Our approach is quite general and it could be applied to a much wider variety of problems. The model reported here was inspired in part by the developmental method for evolving graphs and circuits proposed in (Miller and Thomson, 2003) and particularly by the paper (Khan et al., 2011).

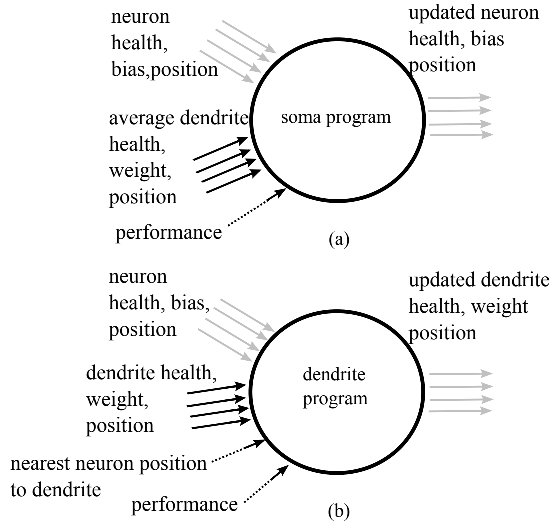
The neuron model

The model is illustrated in in Fig. 1. The neural programs are represented using Cartesian Genetic Programming (CGP) (Miller and Thomson, 2000; Miller, 2011) in which the nodes represent mathematical operations, operating and returning real-values between -1 and 1. The programs read variables associated with neurons and dendrites and produce outputs which are used to update those variables. The dendrite program updates dendrite health, position and weight based on previous values, the parent neuron's health, position and bias and optionally both the position of the nearest neighbouring neuron and a performance score for the whole brain. It should be noted that the performance input is set to zero for non-output neurons. For output neurons, it is set to the performance score corresponding to the problem the output neuron belongs to. When the evolved programs are executed, neurons can move, change, die replicate, grow more dendrites and their dendrites can also change or die. All the neuron and dendrite parameters, weight, bias, health, position are defined by numbers in the range $[-1,-1]$.

The total number of neurons and dendrites possessed by a neuron is bounded between user-defined bounds. This ensures 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, the neuron will be deleted (replicated). Likewise, dendrites are subject to user defined health thresholds which determine whether the dendrite will be deleted or a new one will be created. 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 parameters (δ). If the program output is greater or equal to zero the corresponding neural variable is incremented by the corresponding δ, otherwise it is decremented by the same amount.

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}. Learning epochs allow us to demand that evolution produces a pair of programs that cause the developing ANN to learn. A variable called *incremental training* can be set to choose whether the epochs evaluate a performance score during training (using subsets of the input data) or as a fitness computed over the entire training input. If during training, then the training set grad-

Figure 1: The model of a developmental neuron. Each neuron has a two-dimensional position, health, bias and a variable number of dendrites. Bias refers to an input to the neuron activation function which is added to the weighted sum of inputs (i.e. it is unweighted). Each dendrite has a two dimensional position (the growing tip), a health and a weight. The soma program decides the values of new soma variables position, health and bias based on previous values, and the averages over all dendrites belonging to the neuron of their health, position and weight. Optionally, the dendrite can also use the training performance score for the whole brain (see later).



ually increases until at the last epoch it encompasses the entire training inputs. The training score input is the score obtained by the brain for each problem at the previous epoch (starting with zero). In this scenario, the performance score, since it is an input to the soma and dendrites can influence development during training. Another Boolean user-defined variable called *epoch training* can, if set, require the performance scores to increase at each epoch. In this case, if the performance of the developing network decreases, the epoch learning loop ends and the last improved performance is returned.

Since neurons can move, they can collide with other neurons. If so, the neuron has to be moved until no more collisions take place (collision avoidance). 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. Data arrives at the ANN through inputs at fixed randomly chosen positions between -1 and a user-defined limit. Inputs can be shared or unshared. In the former, the maximum number of inputs required by any of the benchmark problems is determined. If a problem requires less than this then the superfluous inputs provide a zero value. If inputs are unshared then only those inputs relevant to the problem being assessed are presented to the developed ANN.

In our previous work we examined a one-dimensional developmental model and this was applied to multiple classification problems (Miller et al., 2019a). In that work we found that better results were obtained when only output neurons are allowed to move. We also found better results were obtained when neural variables are updated in fixed user-defined increments rather than directly by the evolved programs.

Developing the brain and evaluating the fitness

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

Algorithm 1 Overview of fitness algorithm

1: **function** FITNESS
2: Initialise brain
3: Load 'pre' development parameters
4: **for** NDS_{pre} times **do**
5: Run soma/dendrite programs to update brain
6: **end for**
7: Load 'while' developmental parameters
8: $Epoch = 0$
9: $PreviousFitness = 0$
10: **repeat**
11: **for** NDS_{whi} times **do**
12: Run soma/dendrite programs to update brain
13: **end for**
14: **for** $p = 0$ to $NumBenchmarkProblems$ **do**
15: Extract ANN
16: **if** $IncrementalTraining$ **then**
17: $N_{incr} = (epoch + 1) * NT_{tr}(p)/N_{ep}$
18: **for** N_{incr} training cases **do**
19: Accumulate fitness
20: **end for**
21: $Fitness = Fitness/N_{incr}$
22: **end if**
23: Accumulate Fitness for benchmarks
24: **end for**
25: $Fitness = Fitness/NumBenchmarkProblems$
26: **if** $EpochTraining$ **then**
27: **if** $Fitness < PreviousFitness$ **then**
28: Break
29: **else**
30: $PreviousFitness = Fitness$
31: **end if**
32: **end if**
33: Increment $Epoch$
34: **until** N_{ep} epochs
35: Return $Fitness$
36: **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. Output neurons can change but not die or replicate as the number of output neurons is fixed by the choice of computational problems.

Table 1: Experimental results

Statistic	Experiment A NN position used. Incr. training Performance input No epoch training	Experiment B NN position unused. Incr. training Performance input No epoch training	Experiment C NN position used. No incr. training Performance input Epoch training	Experiment D NN position used. No incr. training No performance input Epoch training
Mean	**0.4072**	0.3357	0.3282	0.3432
Median	**0.2741**	0.2735	0.2732	0.2725
Maximum	**0.7728**	0.7372	0.7703	0.7702
Minimum	0.2701	0.2702	**0.2709**	0.2703
Max Prob 1	0.5458	**0.6699**	0.5459	0.5453
Max Prob 2	**1.0 (4)**	0.9347 (0)	1.0 (1)	1.0 (2)

We extract conventional ANNs from the developed brain in the following way (line 15 in Alg. 1). First, we assign positions to the applied inputs. 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). $NT_{tr}(problem)$ is the number of training cases for each computational problem. If the Boolean variable, $IncrementalTraining$ is set (line 16), then the training cases increase with each epoch, until at the last epoch, the full benchmark training set is evaluated. Thus, development happens during training. If it is not set, then performance is calculated over the the full training set at each epoch.

Experiments and Results

The two reinforcement learning problems we attempt to solve simultaneously are: ball throwing (Koutník et al., 2010; Turner, 2017) and double pole balancing (Turner, 2017). These were chosen because they are established and non-trivial reinforcement learning problems. The ball throwing task is to design a controller which throws a ball as far as possible. The control system has 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 can be determined through simulation and has a value of approximately, 10.202m (Koutník et al., 2010; Turner, 2017).

It is considered solved when the thrown distance greater than or equal to 9.5m (fitness = 0.9312). In double pole balancing, the task is to balance one or more poles on a moveable cart by applying a horizontal force. The inputs to the controller are the position and velocity of the cart and the angle and angular velocity of the pole(s). So there are six inputs. 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 for this number of steps. The fitness for the double pole problem is the fractional number of simulation steps that the poles remain balanced so it is discrete while the fitness for the ball throwing problem is a floating point value. In order to achieve equality between the two objectives the ball throwing fitness was only calculated to two decimal places (this gave better results). In the experiments reported here we allow input sharing, thus for both problems the number of inputs provided to the brain is six, but since the ball throwing problem has only two inputs the remaining values are set to zero.

Twenty evolutionary runs of 20,000 generations of a 1+5-ES were used. The genotype lengths for programs were chosen to be 600 nodes. Goldman mutation (Goldman and Punch, 2015) was used which carries out random point mutation until an active gene is changed. In the experiments reported here we used $N_{ep} = 10$. Clearly this is a small value and investigations are needed to investigate the performance with larger numbers of epochs. Results for four experimental scenarios are presented in Table 1. We investigated whether allowing the dendrite programs to use the position of the nearest neuron was beneficial or not. Experiments A and B differed only in that the position of the

nearest neuron was either supplied (A) to the dendrite program or not (B). In the second pair of experiments (C and D) we disallowed incremental training (so performance was evaluated on the full training set) and used epoch training so that evolution is trying to find programs for the soma and dendrite that mean the performance of the brain improves at each epoch and leads to high fitness.

Experiments A and B support allowing the dendrite program to read the position of its nearest neuron. Furthermore, the superiority of results for experiment A to all other experiments suggests that using incremental training is superior to not using it. Experiments C and D differ only in whether a performance input is given to the neural programs or not. Results suggest that providing performance input or not while using epoch training does not lead to statistically significant differences in results. One evolutionary run in experiment C provided a program that improved performance with each epoch. Further analysis is needed to see if this improvement would continue beyond the maximum epoch (9). The best result has a fitness of 0.7728 (experiment A). In this case, The fitness on the ball throwing task was 0.5456 and the ball was thrown on step 50 (of a possible 10,000 steps) and on the double pole balancing the fitness was 1.0, meaning that for 100,000 time steps the poles were balanced. In other experiments we have occasionally obtained a solution which achieves a perfect score on both problems. When single problems were attempted we found the following results shown in Table 2. Perfect solutions are indicated in parentheses.

Table 2: Results for single problems where the dendrite uses the nearest neuron's positions.

Statistic	Ball throw	Double pole
Mean	0.8515	0.8455
Median	0.9827	1
Maximum	0.9948	1
Minimum	0.5394	0.00278
No. solved	14	12

Single problem results use the same parameters as experiments A of Table 1. They are markedly better than the results for two problems. Of course, solving two problems simultaneously is much harder. However, it appears that often evolution gets stuck on the approximately half-distance solution in ball throwing in which the arm does not swing backwards (which is necessary to achieve long throws).

Conclusion

The developmental model for building neural networks can simultaneously solve two reinforcement learning problems reasonably well, however there still appears to be interference between the networks. The next phase will be to try to understand and eliminate this. Allowing neurons to have a type variable which can be read by the neural programs should help. However, it was investigated in the previous one-dimensional model and shown not to be clearly significant for problem solving (Miller et al., 2019b). Its utility should be investigated further. Also output neurons could be allowed to replicate while still retaining dedicated problem solving output neurons. This might allow more independent behaviour to arise between different problem solving subnetworks.

References

Goldman, B. W. and Punch, W. F. (2015). Analysis of cartesian genetic programmings evolutionary mechanisms. *Evolutionary Computation, IEEE Transactions on*, 19:359 – 373.

Khan, G. M., Miller, J. F., and Halliday, D. M. (2011). Evolution of Cartesian Genetic Programs for Development of Learning Neural Architecture. *Evol. Computation*, 19(3):469–523.

Koutník, J., Gomez, F., and Schmidhüber, J. (2010). Evolving neural networks in compressed weight space. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 619–626.

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

Miller, J. F. and Khan, G. M. (2011). Where is the Brain inside the Brain? *Memetic Computing*, 3(3):217–228.

Miller, J. F. and Thomson, P. (2000). Cartesian genetic programming. In *Proc. European Conf. on Genetic Programming*, volume 10802 of *LNCS*, pages 121–132.

Miller, J. F. and Thomson, P. (2003). A Developmental Method for Growing Graphs and Circuits. In *Proc. Int. Conf. on Evolvable Systems*, volume 2606 of *LNCS*, pages 93–104.

Miller, J. F., Wilson, D. G., and Cussat-Blanc, S. (2019a). Evolving developmental programs that build neural networks for solving multiple problems. In Banzhaf, W., Spector, L., and Sheneman, L., editors, *Genetic Programming Theory and Practice XVI*, chapter 8, pages 137–176. Springer.

Miller, J. F., Wilson, D. G., and Cussat-Blanc, S. (2019b). Evolving programs to build artificial neural networks. In Adamatzky, A. and Kendon, V., editors, *From Astrophysics to Unconventional Computation*, chapter 2, pages 23–71. Springer.

Turner, A. J. (2017). *Evolving Artificial Neural Networks using Cartesian Genetic Programming*. PhD thesis, Department of Electronic Engineering, University of York. Available at `http://etheses.whiterose.ac.uk/12035/`.