Evolving programs that build neural networks for multiple problems

Julian F. Miller[1], Dennis G. Wilson[2], and Sylvain Cussat-Blanc[2]

University of York, Heslington, York, YO10 5DD, UK julian.miller@york.ac.uk
University of Toulouse, IRIT - CNRS - UMR5505, 21 allee de Brienne, Toulouse, France 31015
{dennis.wilson,sylvain.cussat-blanc}@irit.fr

Abstract. Two neural programs are evolved that allow a neuron to build an artificial neural network (ANN) that can perform reasonably well on multiple classification problems simultaneously.

1 Introduction

The original inspiration for ANNs came from knowledge about the brain, however, very few ANN models use evolution and development, both of which are fundamental to the construction of the brain [4]. In this paper, we propose a simple neural model in which two neural programs are required to construct neural networks. One to represent 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 classification 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 [6] and particularly by the paper [3].

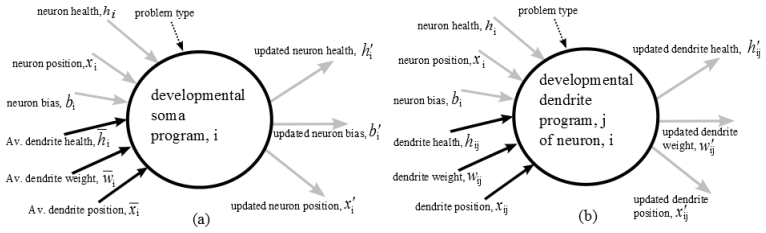
2 The neuron model

The model is illustrated in in Fig. 1. The neural programs are represented using Cartesian Genetic Programming (CGP) [5] using mathematical node operations, operating and returning real-values between -1 and 1. The programs are actually sets of mathematical equations that read variables associated with neurons and dendrites to output updates of those variables. Bias refers to an input to the neuron activation function which is added to the weighted sum of inputs (i.e. it is unweighted).

2.1 Model parameters

The total number of neurons and dendrites possessed by a neuron is bounded between a user-defined bounds. This ensures that a network can not eliminate itself or grow too

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



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 six neural and dendrite variables. The amount of the adjustments are decided by the six user-defined parameters (δ). 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}. In some cases, neurons will collide with other neurons and the neuron has to be moved until no more collisions take place. 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.

2.2 Developing the brain and evaluating the fitness

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

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
```

We extract conventional ANNs from the evolved brain in the following way. First, 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). $N_{ep} > 1$ means that one is looking for a learning algorithm in which the networks improve with iteration.

3 Experiments and Results

We evolve neural programs that build ANNs for solving three classification problems: cancer, diabetes and glass [1]. These were chosen because they are well-studied and also have similar numbers of inputs and a small number of classes. Twenty evolutionary runs of 20,000 generations of a 1+5-ES were used. Genotype lengths for programs were chosen to be 800 nodes. Goldman mutation [2] was used which carries out random point mutation until an active gene is changed.

Experiments were carried out to investigate the utility of neuron movement: all can move, only non output neurons can move, only outputs neurons can move and no neurons can move. We also examined three ways of incrementing or decrementing neural

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

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 delta parameters). In the third, the adjustments to the neural variables are nonlinear (they are adjusted using a sigmoid function). Also, experiments were carried out with $N_{ep} = 10$. Although programs were found that produced ANNs that improved with each epoch, they gradually worsened for more epochs (> 10). We found that the statistically significant best results were obtained when only output neurons were allowed to move. The mean, median, maximum and minimum are shown average over all problems and for each individual problem (cancer, diabetes and glass) in Table 1. Also linear incrementation of neural variables was statistically better than alternatives. These results compared reasonably with huge suite of classification methods as described in [1]. This is encouraging considering that the evolved developmental programs build classifiers for three different classification problems simultaneously.

Table 1. Training and testing accuracy on combined and individual problems when only output neurons are allowed to move.

Acc.	Average Train/Test	Cancer Train (Test)	Diabetes Train (Test)	Glass Train (Test)
Mean	0.7456 (0.7206)	0.9397 (0.9534)	0.7094 (0.6622)	0.5879 (0.5462)
Median	0.7481 (0.7329)	0.9471 (0.9598)	0.7031 (0.6510)	0.5888 (0.5849)
Maximum	0.7854 (0.7740)	0.9657 (0.9942)	0.7526 (0.7500)	0.6636 (0.6415)
Minimum	0.7022 (0.6498)	0.8771 (0.8391)	0.6693 (0.6094)	0.4766 (0.3774)

4 Future work

There are many avenues for future work. Likely short-term plans are to allow the neurons to exist in a two dimensional space and to train evolved networks over a greater variety of computational problems.

References

1. 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)
2. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programmings evolutionary mechanisms. Evolutionary Computation, IEEE Transactions on **19**, 359 – 373 (2015)
3. 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)
4. Miller, J.F., Khan, G.M.: Where is the Brain inside the Brain? Memetic Computing **3**(3), 217–228 (2011)
5. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proc. European Conf. on Genetic Programming. LNCS, vol. 10802, pp. 121–132 (2000)
6. 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)