Evolving developmental neural networks to solve multiple problems

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

Abstract

We describe a neural model in which two evolved neural programs undergo development and form neural networks. The programs decide whether neurons and their dendrites move, change, die or replicate. We show that the programs can build a neural structure from which *multiple* conventional ANNs can be extracted each of which can solve a different computational problem.

Introduction

Although ANNs were originally inspired by the brain (McCulloch and Pitts, 1943), the majority of papers involving neural networks do not use evolution and especially not development. Yet both of these are fundamental to the construction of the brain (Miller and Khan, 2011). Most ANN models obey the "synaptic dogma" of encoding learned knowledge solely in the form of connection strengths (i.e. weights). This gives rise to "catastrophic forgetting" (French, 1999; McCloskey and Cohen, 1989; Ratcliff, 1990). Catastrophic forgetting occurs when an ANN loses it ability to solve an earlier problem when it is retrained on a new one. This is to be expected when the learned information is only encoded in the weights as it is precisely these that are changed when the network is trained. Another problem with the synaptic dogma is that much neuroscience research questions whether memory in brains is even directly related to synaptic strengths since synapses are not fixed structures but are constantly pruned away and replaced by new synapses during learning (Smythies, 2002). There is also a large body of research indicating that learning and environmental interaction are strongly related to *structural* changes in neurons. For instance, animals reared in complex environments where active learning is taking place, have an increased density of dendrites and synapses (Kleim et al., 1998). Indeed, a famous study of London taxi drivers, showed that after their training, their hippocampi were significantly larger relative to those of control subjects (Maguire et al., 2000). Finally, the most significant period of learning in animals happens in infancy, when the brain is developing (Dekaban and Sadowsky, 1978).

The motivation of the work presented in this paper is to find a simplified computational equivalent of the biological neuron. That is a program that allows a neural network to learn for itself, grow as many neurons and connections as it needs, adjust its own weights and biases and solve not only multiple problems but also *unseen* problems. In other words, the long term aim is to find a general AI program. To attempt to achieve this, we propose a neural model which incorporates both evolution and development. In the model, two neural programs acting together construct neural networks. One program represents the neuron soma and the other the dendrite (connection). 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 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. As far as we are aware this is he first attempt in the literature to do this. Our approach is quite general and it could be applied to a much wider variety of problems. In our previous work we examined a one dimensional developmental model and this was applied to multiple classification problems only (Miller et al., 2018, 2019). In this paper we present a two spatial dimensional model and have applied it to up to four computational problems (two classification and two reinforcement learning).

The plan of the paper is as follows. The related work section discusses methods that construct ANNs either by human engineered methods or by development. The following section presents the developmental method for solving multiple computational problems with ANNs. This is followed by a description of two ways of evolving multiple problem solving ANNs: incremental and non-incremental. Next we discuss experimental methodology and results. Finally, we end with a discussion of future work and conclusions.

Related work

Although non-developmental in nature, a few methods have been devised which under supervision gradually augment a fixed architecture of ANNs by adding additional neurons or joining trained ANNs together via extra connections. 'Constructive neural networks' are traditional ANNs which start with a small network and add neurons incrementally while training error is reduced (Fahlman and Lebiere, 1990; Franco and Jerez, 2009). Modular ANNs use multiple ANNs each of which has been trained on a sub-problem and these are combined by a human expert (Sharkey, 2012). More recent approaches adjust weighted connections *between* trained networks on sub-problems guaranteeing that trained networks on sub-problems are unaltered. Rusu et al. applied their method, called 'progressive neural networks' (Rusu et al., 2016) to three classes of problems: variants of the game of Pong, Atari games and 3D maze problems and Terekhov et al. examined their approach on purpose designed image classification tasks (Terekhov et al., 2015). Aljundi et al. have a set of trained ANNs for each task (experts) and use an additional ANN as a recommender as to which expert to use for a particular data instance (Aljundi et al., 2016). They evaluated their approach on image classification tasks and video prediction. Another technique that is related to multiple problem solving is called *transfer learning*. It is a technique that aims to improve a learner in one domain by transferring information from a *related* domain (Pan and Yang, 2010; Weiss et al., 2016). Transfer learning is used particularly when there is a limited supply of target training data. Transfer learning assumes that the data from the various domains are related in some way (i.e. all classification problems, or all images etc.). However, transfer learning does not apply to learning how to solve multiple *unrelated* problems. All these approaches are a form of human engineered development and do not attempt to mimic the way brains are created in the natural world.

More biologically inspired developmental processes (Kumar and Bentley, 2003; Stanley and Miikkulainen, 2003) have been discussed as one of the important processes in an enriched form of artificial evolution called *computational evolution* (Vaario, 1994; Banzhaf et al., 2006). In particular, for several decades authors have investigated various ways of implementing and evolving development processes to construct ANNs. A review of these can be found in (Miller et al., 2019). However, none of the previous research in artificial development has looked at solving standard ANN benchmark problems or multiple problems simultaneously.

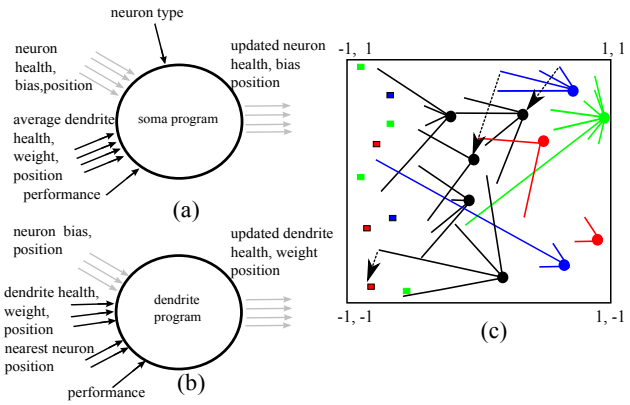
Neuron model

The neural programs are represented using Cartesian Genetic Programming (CGP) Miller and Thomson (2000); Miller (2011, 2020a) in which the program nodes represent mathematical operations, operating on and returning real-values between -1 and 1. CGP was used for two main reasons. Firstly, it allows programs to have an arbitrary number of outputs (unlike tree-based GP), secondly the author is a pioneer in its use. In CGP, 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 preliminary experiments. 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. When the evolved soma and dendrite programs are executed, neurons can move, change, die replicate, grow more dendrites and their dendrites can also change, replicate or die. 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 dendrites are confined to the unit square and all neural variables can only take values between -1.0 and 1.0. There are two kinds of neuron: output and non-output. Every output required by each computational problem has a dedicated *output neuron*. The other neurons are internal and are not directly used to provide outputs from the brain. We refer to these as *non-output neurons*.

Soma program inputs and outputs

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_{nh}, δ_{nb}, δ_{np}. 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. We used hyperbolic tangent for squashing in preference to simple truncation as distinct variables are squashed into distinct values, whereas truncation could map distinct variables to the same value (either

Figure 1: 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. Neural programs read these variables and update them (a and b). When the two programs are executed a collection of neurons are created (c). External inputs for different computational problems are shown as red, green or blue coloured rectangles. Output neurons for different problems have corresponding colours. There are assumed to be two outputs each for the red and blue problems and one for the green. Black neurons are non-output. Dashed arrows show dendrites forming connection with nearest neurons or inputs on the left (snapping) to achieve connected networks.



1 or -1). 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.

Dendrite program inputs and outputs

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 initialized 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.

Development happens in two distinct phases ('pre' learning and 'while' learning). 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 5-7). While in the 'while' phase the brain is assessed and provides feedback to the developmental process (lines 10-12).

Lines 9 - 29 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 demand that evolution produce a pair of programs that cause the developing ANN to learn. The learning loop only continues while the training accuracy does not decrease (lines 24-27). If it does, the algorithm stops and returns the fitness at 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 given epoch over all benchmarks is given as the performance input. 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). Firstly, only the inputs *relevant* to the target problem in hand are given their randomly pre-assigned positions. We do not have input neurons in the model, rather input data is supplied at fixed spatial locations unique to each problem. These can be thought of as electrodes, supplying current. In this way, the brain is only supplied with relevant inputs and can not connect to irrelevant inputs (for another target problem). In earlier work, we allowed all target problems to be present and we assigned zero values to any inputs that were irrelevant to the target problem in hand. This has a big drawback as it means that the brain could be flooded with many irrelevant inputs having the value zero. Biological brains appear to have a sophisticated method of ignoring irrelevant inputs, however it is not yet clear how this happens. There are output neurons dedicated to each problem these are free to move around as

Algorithm 1 Fitness algorithm.

```
function FITNESS
    Initialise brain
    Load 'pre' development parameters
    PrevFitness = 0
    for NDS_pre times do
        Run soma/dendrite programs to update brain
    end for
    Load 'while' developmental parameters
    for epoch = 1 to N_ep do
        for NDS_whi times do
            Run evolved programs to update brain
        end for
        TotFitness = 0
        for p = 1 to NumBenchmarkProblems do
            Extract ANN
            Fitness(p) = 0
            for NT(p) training cases do
                Fitness(p) = Fitness(p) + FitInstance
            end for
            Fitness(p) = Fitness(p)/NT(p)
            TotFitness = TotFitness + Fitness(p)
        end for
        TotFitness = TotFitness/NumBenchmarkProblems
        if TotFitness < PrevFitness then
            Break
        else
            PrevFitness = TotFitness
        end if
    end for
    Return TotFitness
end function
```

directed by the soma program, how they move is dictated by their inputs. There is no input supplied to the soma and dendrite program that indicates what problem is being solved (i.e. like a problem ID).

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 (to avoid recurrence). 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 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). $NT(p)$ 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. Other activation functions can be easily be used, however in experiments so far, the hyperbolic tangent appears to perform the best.

Model parameters

Table 1: Table of neural model parameters.

Parameter	Value
$NN_{min}(NN_{max})$	0 (30)
N_{init}	5
$DN_{min}(DN_{max})$	1 (60)
ND_{init}	5
N_{ep}	8
MN_{inc}	0.0001
I_u	-0.6
O_l	1.0
α	1.5
'Pre' development parameters	
NDS_{pre}	6
θ_{nd} (θ_{nb})	-0.6 (0.2)
θ_{dd} (θ_{db})	-0.7 (0.2)
δ_{nh} and δ_{dh}	0.2
Remaining δs	0.1
'While' development parameters	
NDS_{whi}	1
θ_{nd} (θ_{nb})	-0.4 (0.2)
θ_{dd} (θ_{db})	-0.65 (-0.6)
All δs	0.1

The model has a large number of user-defined parameters these are shown in Table 1. It is hoped that as the model develops some can be removed or assume default values, however, at present the approach has been to create a neuron model that is as general as possible and with the least number of assumptions. The initial number of non-output neurons, can be chosen by the user and is denoted, N_{init}. 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.

If the health of a neuron falls below (exceeds) a user-defined threshold, θ_{nd} (θ_{nb}) the neuron will be deleted (repli-

cated). Likewise, dendrites are subject to user defined health thresholds, θ_{dd} (θ_{db}) 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 θ_{db}. This choice was made to prevent the potentially very rapid growth of dendrite numbers.

Neurons that have just been born are placed above and to the right of the parent neuron, by adding a small increment, MN_{inc} to the parent's x and y position. Newly born neurons are given a health equal to one, a bias of zero, and the same number of dendrites as the parent. Their dendrites are given weight and health equal to zero. The x and y positions of the dendrites are set to zero. When individual dendrites are created they are given a weight and health equal to one and their x and y-positions are set to 0.8 of the parent x and y-position. In preliminary empirical investigations these choices were found to work well. 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 x positions of data inputs to the brain are given fixed random values between -1 and -1+I_u while the y-positions take randomly chosen values between -1 and 1. The output neurons for all problems are initially placed at x-position O_l and their positions on the y-axis are uniformly distributed between -1 and 1. However, 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. When solving a particular problem, output data is read from only those output neurons corresponding to the chosen problem (the remaining output neurons are ignored). Note, non-output neurons are not allowed to connect to output neurons and output neurons can only connect to non-output neurons or inputs. When output neurons are initialized their health and bias are given random values between -1 and 1. All neurons are initialized with ND_{init} dendrites. The dendrites variables, health and weight are initialized with random values between -1 and 1. They are given randomly chosen x and y-positions between 0 and 1. Finally, the neural activation function has a slope constant given by α. The chosen experimental parameters for this study are shown in Table 1. After much experimentation these were found to work quite well. The values found for $DH_{death} = -0.65$ and $DH_{birth} = -0.6$ are quite surprising as learning is taking place while there is a high probability of dendritic change (birth or death).

Incremental evolutionary problem solving

The user can choose either to evolve developmental programs that solve all problems at the same time or evolve solutions incrementally until on the last round of evolution all problems are solved simultaneously. We refer to the latter as *incremental problem solving* (IPS). In non-incremental problem solving a performance score calculated for each benchmark. The fitness is then the average of the performances. Incremental problem solving works as follows. First the programs are evolved for a number of generations to solve a single benchmark problem, then when either the user-chosen generation limit has been reached or a perfect score has been obtained, evolution tries to evolve programs that solve two benchmarks. This process repeats until at the last evolutionary cycle all benchmarks are attempted. The thinking here is that rather than trying to evolve soma and dendrite programs that are always evaluated on all benchmarks, the incremental problem solving method allows evolution to create programs that perform well on problems individually before moving on to more problems. It is important to note that the final evolved programs after both non-incremental and incremental problem solving merely have to be executed (i.e. without evolution) to perform on all the benchmark problems. The hope is that after a sufficient number of computational problems, evolution might discover general programs that construct an autonomous learning system. The role of the reward signal is important for general problem solving as it provides environmental feedback to the developing artificial brain so that it can, if necessary, change its behaviour towards more useful neural changes.

Experimental Methodology

In this work, we look to simultaneously solve two standard classification problems (diabetes and glass) and two reinforcement learning problems (double-pole balancing and ball throwing). The definitions of these problems are available in the UCI repository [1]. Diabetes has 8 real attributes and two Boolean outputs. Glass has 9 real attributes and six Boolean outputs. The specific datsets chosen were diabetes1 and glass1 which are described in the PROBEN suite of problems [2]. The two reinforcement learning problems are ball throwing (Koutník et al., 2010; Turner, 2017) and double pole balancing (Turner, 2017). The ball throwing 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 can be determined through simulation and has a value of approximately, 10.202m. Ball throwing is considered solved when the thrown distance greater than or equal to 9.5m (fitness = 0.9312). It has a strong sub-optimal fitness

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

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

value where the fitness is 0.546 this corresponds to the maximum possible distance that the ball can be thrown when the arm only swings forward, whereas to achieve maximum distance, one needs to swing the arm backwards so that it picks up speed due to gravity before torque is applied. In double pole balancing, the task is to balance two poles on a moveable cart on a limited track by applying a horizontal force to the cart. 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 to within certain limits 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 the fitness is fractional while the fitness for the ball throwing problem is a continuous floating point value.

Experiments and results

The genotype length was 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. When incremental problem solving is used 20,000 generations are used for each problem combination. For example, solving problem 1 is given 20,000 generations, solving both problem 1 and 2 is given another 20,000 generations and so on. When comparisons are made with non-incremental learning we allow the latter to have the same *total* number of generations. In all experiments twenty evolutionary runs of a 1+5-ES were used. All experiments used exactly the same parameters as shown in Table 1.

Single problems

Initially, to assess baseline performance we conducted experiments on single benchmark problems. Of course, incremental problem solving does not apply since there is only one problem. Results are presented in Table 2.

As can be seen in Table 2 17 out of 20 runs produced soma and dendrite programs that can build ANNs that can completely balance two poles. While 16 out of 20 runs on the ball throwing problem produced programs that could build ANNS with thrown distances above 9.5m (classed as solved). In the case of classification we provide both the fitness scores (accuracy) and the accuracy on unseen test sets (shown in parentheses). For comparison we have included results for solving single classification problems of 179 classifiers (covering 17 families of ML methods) (Fernández-Delgado et al., 2014). However one should be cautious comparing the model results with other machine learning as these methods are not developmental in nature. Also, there has been no prior work on developmental approaches on standard ANN benchmark problems.

Table 2: Results for single problems.

Statistic	Double pole	Ball throw
Mean	0.9559	0.8961
Median	1	0.9891
Maximum	1	0.9990
Minimum	0.6129	0.5460
No. solved	17	16
	Glass	Diabetes
	train (test)	train (test)
Mean	0.6360 (0.530)	0.7542 (0.6961)
Median	0.6449 (0.5660)	0.7500 (0.6745)
Maximum	0.6916 (0.6415)	0.7865 (0.7604)
Minimum	0.5514 (0.0943)	0.7396 (0.6510)
ML methods	Glass test	Diabetes test
Mean	0.610	0.743
Maximum	0.785	0.790
Minimum	0.319	0.582

Incremental v. non-incremental problem solving

In the next series of experiments we compared the effectiveness of solving pairs of problems incrementally (IPS) with solving them non-incrementally. Table 3 shows the results. In incremental problem solving 20,000 generations of evolution are carried out for each problem. In non-incremental, the total number of generations is 20,000 multiplied by the number of problems. So, the two scenarios use the same total number of generations. It is clear that it is more effective to develop an ANN that solves one problem first and then alter or augment it to solve, in addition, another different problem. Observing the non-incremental runs showed many occasions where improvements in fitness for the double pole task were lost when improvements occurred in the fitness for the ball throwing problem. The success of incremental problem solving indicates that it is less likely that previous learned high performing neural networks will be destroyed (by forgetting) when another problem is being solved. However, occasionally the achieved fitness on the first problem worsened briefly when there were large improvement in the second problem, indicating that interference still occurs.

The issue of interference can be studied by examining the performance when solving a single problem and comparing it with the performance in solving the same problem while at the same time solving another problem. Table 4 compares the performance when solving either the ball throw problem or diabetes classification only, to the respective performance while maintaining an solution to double pole balancing or glass classification. If there was no interference, one would expect the performances in the two cases to be the same as their respective single problems. The best solution for double pole and ball throw achieved a perfect score for double pole and the score of 0.5385 for the ball throwing problem.

Table 3: Incremental problem solving versus non-incremental problem solving.

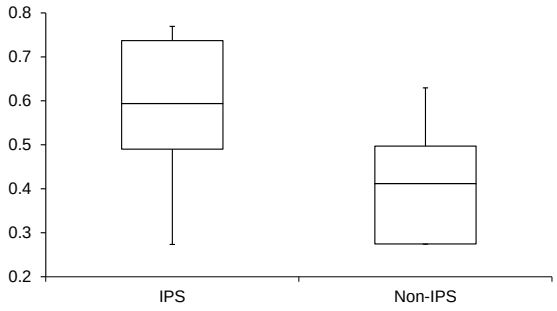
Statistic	Incremental Problem solving	Non-incremental Problem solving
	double pole (DP) ball throwing (BT)	double pole ball throwing
Mean	**0.5844**	0.4002
Median	**0.6050**	0.4449
Maximum	**0.7693**	0.6294
Minimum	0.2735	**0.2742**
No. solved DP	**8**	1
No. solved BT	1	**9**
	Glass diabetes train (test)	Glass diabetes train (test)
Mean	**0.6644 (0.6183)**	0.6561 (0.6009)
Median	**0.6599 (0.6274)**	0.6566 (0.5999)
Maximum	**0.7029 (0.6974)**	0.6948 (0.6502)
Minimum	**0.6342** (0.5158)	0.6015 (**0.5347**)

The best solution in the paired classification case was an accuracy of 0.6635 for glass and 0.7422 for diabetes. The best average classification accuracy on the test data sets was 0.6974. It had an accuracy of 0.6604 for glass and 0.7344 for diabetes. This is quite respectable since the average test accuracy of 179 ML techniques has an average test accuracy of 0.610 for glass and 0.743 for diabetes. For visualization box plots are shown in Fig. 2. Only maximum or minimum outliers are shown.

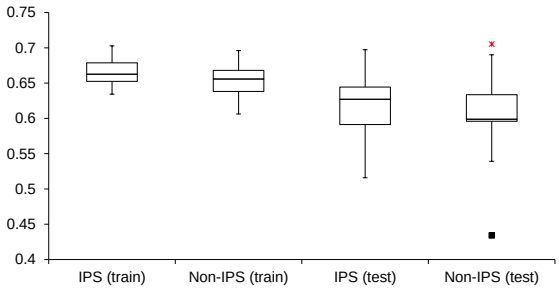
Table 4: Solving one problem while maintaining a previously evolved solution compared with just solving a single problem.

Statistic	Ball throwing fitness while solving DP	Ball throwing only
Mean	0.3978	**0.8961**
Median	0.5020	**0.9891**
Maximum	0.9800	**0.9990**
Minimum	0.0000	**0.5460**
No. solved DP	8	**16**
No. solved BT	1	-
	Diabetes fitness while solving glass	Diabetes only
Mean	0.7325 (**0.7224**)	**0.7542** (0.6961)
Median	0.7422 (**0.7344**)	**0.7500** (0.6745)
Maximum	0.7656 (**0.7656**)	**0.7865** (0.7604)
Minimum	0.6693 (0.6354)	**0.7396(0.6510)**

Clearly much better results are obtained when solving the ball throwing task alone than trying to solve it while maintaining a brain that can solve the double pole problem. This indicates that interference is still occurring. The developed classifier for diabetes which also classifies glass is a little worse than the developed classifier for diabetes alone. However it appears that there is less interference with classifiers than between ball throwing and double pole balancing. It is interesting to note that the combined classifier generalizes better than the classifier for diabetes alone.



(a) Double pole balancing and ball throwing



(b) Glass and diabetes classification

Figure 2: Performance of Incremental problem solving (IPS) versus non-incremental problem solving

Four problems

In the next series of experiments we attempted to solve four problems simultaneously. We do this incrementally, first

double pole balancing, then ball throwing, followed by glass and diabetes classification. Table 5 shows the results. The figures in the table refer to the final fitness after solving all four problems simultaneously. The fittest solution had a fitness of 0.7689 and the problem scores that contribute to that are shown in the table. The average number of nodes in the CGP program graphs for both the soma and dendrite (over 20 evolutionary runs) was 55. So even though the maximum allowed size of these is 600 nodes a much smaller number are active. This is quite typical for CGP. To summarize, when the evolved programs for the soma and dendrite were executed over two epochs the developed brain can solve the pole balancing problem and the arm throw and at the same time obtain the test classification accuracies of 0.4340 for glass and 0.3646 for diabetes. Note, that the total number of generations that programs were evolved attempting problem $p = 1$ to 4 is $20{,}000 \times p$. Thus for problem 1 (DP) the total number of generation was 80,000 since it tries to solve DP all the time, while for BT it was 60,000 as it only starts trying to solve BT after 20,000 generations have elapsed trying to solve DP alone. This explains why results appear to be better for earlier problems.

Pole balancing used 4 non-output neurons. One of these was a "zero neuron" whose bias was zero and its fifteen dendrites all had weight zero! Ball throwing also used four non-output neurons but three were shared with other problems. Glass used eight non-output neurons four are shared with other problems. The other four were zero neurons with 15 dendrites. The three non-output neurons of diabetes were all shared. Note, neurons that are shared in the brain can have different connections in the extracted ANNs due to the snapping mechanism. The total number of distinct non-output neurons used in the developed brain was only eight (including four zero producing neurons). Of course, there are also eleven output neurons required by the benchmark problems.

Table 5: Solving four problems.

Statistic	Pole balancing	Ball throwing
Mean	0.9039	0.4930
Median	1.0	0.5285
Maximum	1.0	0.9673
Minimum	0.0001	0.0
No. solved	14	1
Fittest	1	0.9673
	Glass train (test)	Diabetes train (test)
Mean	0.4907 (0.4000)	0.6717 (0.6271)
Median	0.4766 (0.3962)	0.6719 (0.6406)
Maximum	0.5888 (0.5660)	0.6979 (0.6927)
Minimum	0.4206 (0.1321)	0.6354 (0.3646)
Fittest	0.4393 (0.4340)	0.6693 (0.3646)

Conclusion and Future Work

Although we were able to evolve a computational brain that can solve multiple machine learning problems reasonably well, our attempts were hindered by interference. How does natural evolution find improvements in the performance of systems without the deterioration of already evolved systems? A much larger numbers of neurons and dendrites may make it easier for evolution. Achieving non-interfering development may simply take more evolutionary time. Allowing output neurons dedicated to particular problems to replicate (currently disallowed) and handling the issue of where to collect the output for a particular problem is one possible approach. It would be interesting to determine whether memory is predominantly attributable to weights or soma and dendrite positions. Further analysis of the evolved neural programs is required to understand how the networks learn.

In brains, electrical signals passing between neurons influences dendrite or axon growth and shapes brain morphology. This is called activity dependence (Ooyen, 2003). Activity dependence can be incorporated into our neural model. For instance, the health, weight or position of dendrites could be incremented or decremented depending on the strength of signal passing along the dendrite (i.e. via thresholding). Weights could be adjusted in a non-temporal Hebbian way, where if a high signal passed along a dendrite and the parent neuron also outputted a high (low) signal, then dendrite weight could increase (decrease). Similarly neuron properties could be influenced by the neuron's output signal. These ideas have been implemented (Miller, 2020b).

The long-term aim of this work is to evolved developmental programs that create a self-learning neural system. The idea behind epoch learning was to allow development to take place until the brain stopped improving. The hope is that by evolving developmental programs over sufficient numbers of epochs would encourage generality and result in a self-improving brain. A first step would be to see if evolved programs that solve a number of problems of a particular type (i.e. classification) would be able to solve *unseen problems* of the same type.

We have evaluated our approach on standard benchmarks in machine learning. However, it might be better to create new suites of simpler problems for developmental methods (e.g. object recognition) where one can start with simple problems and gradually increase the task complexity.

References

Aljundi, R., Chakravarty, P., and Tuytelaars, T. (2016). Expert gate: Lifelong learning with a network of experts. *CoRR, abs/1611.06194*, 2.

Banzhaf, W., Beslon, G., Christensen, S., Foster, J. A., Képès, F., Lefort, V., Miller, J. F., Radman, M., and

Ramsden, J. J. (2006). From artificial evolution to computational evolution: a research agenda. *Nature Reviews Genetics*, 7:729–735.

Dekaban, A. S. and Sadowsky, D. (1978). Changes in brain weights during the span of human life. *Ann. Neurol.*, 4:345–356.

Fahlman, S. E. and Lebiere, C. (1990). The cascade-correlation learning architecture. In *Advances in neural information processing systems*, pages 524–532.

Fernández-Delgado, M., Cernadas, E., Barro, S., and Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.*, 15(1):3133–3181.

Franco, L. and Jerez, J. M. (2009). *Constructive neural networks*, volume 258. Springer.

French, R. M. (1999). Catastrophic Forgetting in Connectionist Networks: Causes, Consequences and Solutions. *Trends in Cognitive Sciences*, 3(4):128–135.

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

Kleim, J., Napper, R., Swain, R., Armstrong, K., Jones, T., and Greenough, W. (1998). Selective synaptic plasticity within the cerebellar cortex following complex motor skill learning. *Neurobiology of Learning and Memory*, 69:274–289.

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

Kumar, S. and Bentley, P., editors (2003). *On Growth, Form and Computers*. Academic Press.

Maguire, E. A., Gadian, D. G., Johnsrude, I. S., Good, C. D., Ashburner, J., Frackowiak, R. S. J., and Frith, C. D. (2000). Navigation-related structural change in the hippocampi of taxi drivers. *PNAS*, 97:4398–4403.

McCloskey, M. and Cohen, N. (1989). Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *The Psychology of Learning and Motivation*, 24:109–165.

McCulloch and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133.

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

Miller, J. F. (2020a). Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*, 21:129–168.

Miller, J. F. (2020b). Multiple problem solving with evolved developmental neural networks: activity dependence. In *3rd workshop on Developmental Neural Networks (DevANN2020) at ALIFE2020*.

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., Wilson, D. G., and Cussat-Blanc, S. (2018). Evolving developmental programs that build neural networks for solving multiple problems. In *Genetic Programming Theory and Practice XVI*, pages 137–178.

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.

Ooyen, A. V., editor (2003). *Modeling Neural Development*. MIT Press.

Pan, S. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1354–1359.

Ratcliff, R. (1990). Connectionist Models of Recognition and Memory: Constraints Imposed by Learning and Forgetting Functions. *Psychological Review*, 97:205–308.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671*.

Sharkey, A. J. (2012). *Combining artificial neural nets: ensemble and modular multi-net systems*. Springer Science and Business Media.

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

Stanley, K. O. and Miikkulainen, R. (2003). A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130.

Terekhov, A. V., Montone, G., and OŔegan, J. K. (2015). Knowledge transfer in deep block-modular neural networks. In *Conference on Biomimetic and Biohybrid Systems*, pages 268–279. 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/`.

Vaario, J. (1994). From evolutionary computation to computational evolution. *Informatica*, 18:417–434.

Weiss, K., Khoshgoftaar, T., and Wang, D. (2016). A survey of transfer learning. *Journal of Big Data*, 3(1):9.