DOI: http://dx.doi.org/10.7551/978-0-262-32621-6-ch108

Evolving Autonomous Agent Controllers as Analytical Mathematical Models

Paul Grouchy[1] and Gabriele M.T. D'Eleuterio[1]

[1]University of Toronto Institute for Aerospace Studies, Toronto, Ontario, Canada M3H 5T6
paul.grouchy@mail.utoronto.ca

Abstract

A novel Artificial Life paradigm is proposed where autonomous agents are controlled via genetically-encoded Evolvable Mathematical Models (EMMs). Agent/environment inputs are mapped to agent outputs via equation trees which are evolved using Genetic Programming. Equations use only the four basic mathematical operators: addition, subtraction, multiplication and division. Experiments on the discrete Double-T Maze with Homing problem are performed; the source code has been made available. Results demonstrate that autonomous controllers with learning capabilities can be evolved as analytical mathematical models of behavior, and that neuroplasticity and neuromodulation can emerge within this paradigm without having these special functionalities specified *a priori*.

Introduction

When looking to design an Artificial Intelligence (AI) capable of robust and adaptable behavior, one must first choose how to represent such an agent *in silico*. Although many such representations exist, two of the most common are direct computer code representations, which can be programmed autonomously via Genetic Programming (GP) (Koza, 1992; Poli et al., 2008), and Artificial Neural Networks (ANNs), which are computational models based on the biological neural networks of animal brains that can be designed using one of a variety of approaches (Floreano et al., 2008).

Both of these representations require significant *a priori* design decisions. For a GP approach, one must choose which programming operations to include (e.g., bit-shift, and, or, xor, if-then-else, etc.), with a trade-off between potential program capabilities and search space size (and thus the ability of GP to find a solution). With ANNs, one must choose from many different types (e.g., Feedforward, Recurrent, Continuous-Time, Spiking, etc.), with trade-offs that have not yet been systematically investigated (Floreano et al., 2008). Other important design decisions for ANNs include the type of activation function and the incorporation of learning capabilities. Thus, to arrive at an agent representation appropriate for their task, a designer must first invest a significant amount of time and computational resources. Furthermore, any such design decisions introduce additional experimenter bias into the simulation. Finally, complex behaviors require complex representations, adding to the already high computational costs of ALife algorithms while further obfuscating the inner workings of the evolved agents.

We propose Evolvable Mathematical Models (EMMs), a novel paradigm whereby autonomous agents are evolved via GP as analytical mathematical models of behavior. Agent inputs are mapped to outputs via a system of genetically encoded equations. Only the four basic mathematical operations (addition, subtraction, multiplication and division) are required, as they can be used to approximate any analytic function, thus eliminating the need to determine which operators to employ. More sophisticated operations can of course be added at the will of the designer; however, theoretically the four basic ones constitute the minimal set of operations required and accordingly the designer can be relieved of much of the guessing game. Furthermore, since agents are represented directly as mathematical equations, the evolved agents are amenable to mathematical analysis *post facto*.

The rest of this paper is organized as follows. In the next section, previous work on evolving ANNs with learning behaviors and on Genetic Programming for agent control is presented. This is followed by the algorithmic details of EMMs. A Double-T Maze problem that requires learning capabilities is then described, with the results of EMM experiments in this domain presented in the subsequent section. The source code for these experiments is provided. Finally, conclusions and future work will be discussed.

Background

Neuroplasticity and Neuromodulation

Autonomous agents are often evolved as artificial neural networks (ANNs). Typically, when one is evaluating an ANN, either in simulation or hardware, the ANN's connection weights are fixed. It is only when generating offspring genomes that genetic operators such as mutation can modify the ANN's weights. This is contrary to biological neural

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

networks, however, where "neuroplasticity" allows for connections between neurons to change during the lifespan of an organism (Pascual-Leone et al., 2005). Neuroplasticity (or "plasticity" for short) is what enables biological organisms to learn, modifying the way they react to certain inputs from the environment.

Hebbian learning, which is based on how learning is thought to occur in biology (Hebb, 1949), can be used to implement plasticity in ANNs (Floreano and Mondada, 1996). In this paradigm, local learning rules Δw_{ij} are evolved for each network connection weight w_{ij}. After each timestep t of an ANN's lifespan, each of its connection weights is updated using its associated learning rule

$$w_{ij}^{t+\Delta t} = w_{ij}^{t} + \eta \Delta w_{ij}^{t+\Delta t} , \quad (1)$$

where $0 \leq \eta \leq 1$ is the evolvable learning rate. An example learning rule Δw_{ij} is the "plain Hebb rule," defined as

$$\Delta w_{ij}^{t+\Delta t} = (1 - w_{ij}^{t}) v_i^t v_j^t , \quad (2)$$

where v_i^t is the output of neuron i at the current timestep.

In the learning paradigm described above, connection weights are adjusted at every timestep throughout the lifetime of an agent. This differs from biological systems which are theorized to use "neuromodulation" to control and stabilize learning (Bailey et al., 2000).

To improve learning algorithm performance, the Analog Genetic Encoding (AGE) algorithm, an encoding method based on biological gene regulatory networks that can evolve both the connections weights and structure of an ANN (Mattiussi and Floreano, 2004; Dürr et al., 2006), was modified to allow for Hebbian learning which could be enabled and disabled via neuromodulatory signals (Soltoggio et al., 2007; Durr et al., 2008; Soltoggio et al., 2008). In this approach, a generalized Hebbian rule from (Niv et al., 2002) was modified to include a modulatory signal m:

$$\Delta w_{ij}^{t+\Delta t} = m^t \eta \left(A v_i^t v_j^t + B v_i^t + C v_j^t + D \right) \quad (3)$$

where A, B, C and D are evolvable parameters that determine the importance of the different types of Hebbian learning and $0 \leq m^t \leq 1$ is the current strength of the signal produced by one or more special modulatory neurons. These modulatory neurons operate in a manner similar to regular neurons, taking inputs from other neurons in the network through network connections and generating their output value m using an activation function.

Fixed-Weight Learning

Learning behaviors have been observed in several fixed-weight ANN-based experiments as well. Fixed-weight recurrent neural networks seem to be able to accomplish certain tasks requiring learning, as the recurrent neural connections can act as a type of memory (e.g., (Stanley et al., 2003; Soltoggio et al., 2008)). Continuous-time recurrent neural networks (CTRNNs) have also demonstrated learning capabilities, sometimes even outperforming plastic neural networks (Jesper and Floreano, 2002; Tuci and Quinn, 2003).

Genetic Programming for Agent Control

Genetic Programming (GP) can be used to evolve autonomous controllers as computer programs. In the first such experiment (Koza and Rice, 1992), controllers were represented as variable-length trees containing sensor inputs and four preprogrammed macros, such as "if-less-than-or-equal." Programs that could control a simulated robot to find a box in an irregularly shaped world and push it to a wall from four different starting configurations were evolved.

A linear implementation of GP was used to evolve machine code to control a Khepera robot in (Nordin and Banzhaf, 1995, 1997). This work was recently extended in (Burbidge et al., 2009; Burbidge and Wilson, 2014), where machine code was evolved using Grammatical Evolution. Here, agent genomes are binary strings that are mapped to their machine code phenotypes via a prespecified generative grammar.

GP has also been used to evolve competitors for the RoboCup robotic soccer tournament. The team "Darwin United" was evolved using GP with a variety of possible operations, including basic mathematical operators, reading and writing to memory locations and executing a variety of programmer-designed subroutines (Andre and Teller, 1999). A simple GP approach was also used to evolve robot goalie behaviors in (Adorni et al., 1999). This work is the most similar to our Evolvable Mathematical Models paradigm presented below, as it is evolving mathematical equations as trees for robot control. However, these experiments were performed on a relatively simple task (especially considering the amount of information provided to the controller), had only one equation tree/agent output, employed two additional operators (sine and cosine), and did not allow for extra state variables/equation trees to be evolved.

Evolvable Mathematical Models

Our Evolvable Mathematical Models (EMM) algorithm uses Genetic Programming (GP) to evolve mathematical models of behavior. An earlier version of this algorithm that evolved one ODE per agent output (i.e., there were no extra state variables) was presented in (Grouchy and D'Eleuterio, 2010) and implemented in (Grouchy and Lipson, 2012). The core idea is that one can evolve autonomous agent controllers as mathematical equations that map from agent inputs to outputs.

Representation

An EMM-based agent is represented as a system of equations, with one equation for each of the N experimenter-defined outputs v_i in the simulation. Additional "extra"

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

equations can be added through an “add-equation” mutation (similar to the concepts of Automatically Defined Functions and Architecture-Altering Operations (Koza, 1994)). An agent's N' extra equations do not have associated agent outputs; however, they modify agent outputs indirectly via their incorporation into those equations that affect agent outputs directly. Therefore, an agent is fully specified by its system of state equations

$$\mathbf{v}^{t+\Delta t} = \mathbf{f}\left(\mathbf{u}^t, \mathbf{v}^t\right) \tag{4}$$

and its evolvable initial conditions $\mathbf{v}^{t=0}$. Here,

$$\mathbf{v} = \left[v_1, v_2, ..., v_N, v_{N+1}, ..., v_{N+N'}\right]^{\mathrm{T}} \tag{5}$$

and

$$\mathbf{u} = \left[u_1, u_2, ..., u_M\right]^{\mathrm{T}} , \tag{6}$$

where M is the number of experimenter defined inputs to the agent and

$$\mathbf{f}\left(\mathbf{u}, \mathbf{v}\right) = \left[f_1\left(\mathbf{u}, \mathbf{v}\right), f_2\left(\mathbf{u}, \mathbf{v}\right), ..., f_{N+N'}\left(\mathbf{u}, \mathbf{v}\right)\right]^{\mathrm{T}} \tag{7}$$

are the agent's genetically encoded state functions.

For all experiments presented here, tree structures were used to represent the EMMs in the agent genome, as in canonical GP. Only the four basic mathematical operators are allowed (addition, subtraction, multiplication and division), from which any analytic function can be approximated. A real-valued variable-length vector was used to represent an agent's initial conditions $\mathbf{v}^{t=0}$ in the genome.

A genome contains $N + N'$ equation trees, one for each output and extra state variable of the system. Each tree contains a collection of terminal and nonterminal nodes. The set of possible terminal nodes is comprised of all possible real constants and variables (i.e., input, output and extra state), while the set of possible nonterminal nodes is composed of addition, subtraction, multiplication and division. The number of “child” nodes (subtrees) of a nonterminal node is two, as all four of the basic operations have an arity of two. An example genome for an agent with $N + N' = 2$ is shown in Figure 1.

Evolution

Initialization. The random initialization of the N initial equation trees in each initial genome (e.g., at generation 0 in a genetic algorithm) is done using the “ramped half-and-half” method from GP. This method is a combination of two methods, the “full” and “grow” methods. For both methods, a maximum depth (i.e., the maximum number of edges that need to be traversed to reach a node, starting from the root node) is specified. In the “full” method, nonterminal (i.e., operator) nodes are randomly generated until the maximum depth is reached. At the maximum depth, only terminal (i.e., operand) nodes are created. In the “grow” method,

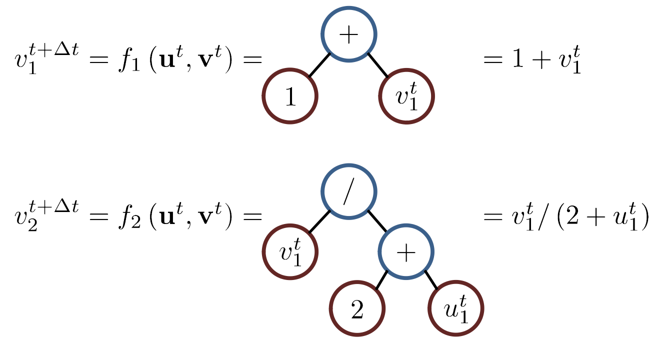

Figure 1: An example EMM for an agent with $N + N' = 2$. The two trees, along with the two initial values $v_1^{t=0}$ and $v_2^{t=0}$ (not shown), are how the agent's EMM is encoded in its genome.

as in the “full” method, only terminal nodes are created at the maximum depth. The difference is that before the maximum depth is reached, randomly generated nodes can be either terminal or nonterminal nodes (with equal probability), allowing for a wider range of potential tree shapes. For all experiments presented here, half of the trees are generated with a maximum depth of 1, while the other half have a maximum depth of 2. Terminal nodes are set to a randomly chosen variable or a random constant with equal probability. Initial genomes do not contain any extra state variables, i.e., $N' = 0$. Initially, constants are randomly selected and initial output values $\mathbf{v}^{t=0}$ are set to random values.

Sexual Recombination. Sexual recombination allows for large jumps in the search space through combining two partial solutions. Any such genetic operation requires two parents to produce an offspring genome. Otherwise, a single parent's genome is cloned to produce an offspring genome. In EMMs, tree-level sexual recombination occurs in a fashion similar to crossover in GP.

An offspring genome is initially generated as a clone of the first parent. If a tree in the offspring genome is selected to undergo tree-level recombination, one of its nodes is selected at random and replaced with a randomly selected subtree from the second parent. Randomly selected nodes have a probability of 0.1 of being terminal nodes. If there are subtrees below the selected node in the offspring's tree, they are discarded. The subtree from the second parent can come from any of its equation trees. This allows for partial solutions to be reused and to be copied to different equation trees. This subtree grafting operation is similar to the subtree mutation operation described below. If the subtree being copied over references extra state variables that are not present in the offspring genome, the equations for those state variables are copied to the offspring genome from the second parent. An offspring genome is subject to a variety of

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

genetic mutations, even if it has been produced via sexual recombination.

Tree Mutation. Tree mutations (as well as extra state variable mutations and sexual recombination) can occur when a parent genome is being copied to its offspring. This means that mutations can only occur *between* generations. Agent genomes remain fixed throughout an agent's lifetime.

These mutations are implemented in a manner similar to GP. If a tree is selected to be mutated, one of a variety of mutations is applied:

- **Point Mutation.** A point mutation performs one of several operations on a single randomly selected node in the tree:
  - *Perturbation of a constant.* This operation can only be performed if the tree in question contains one or more constants. The operation adds a random value to a randomly selected constant.
  - *Mutation of a nonterminal node.* This operation is performed if a perturbation of a constant was not done and if a randomly selected node is a nonterminal. A new nonterminal operation is randomly selected from the set of addition, subtraction, multiplication and division.
  - *Mutation of a terminal node.* This operation is performed if a perturbation of a constant was not done and if a randomly selected node is a terminal. One of two mutations occurs, with equal probability. The chosen terminal is either mutated to a randomly chosen variable or it is mutated to a random constant.
- **Subtree Mutation.** This operation selects a random node on the original tree and replaces it with a new random subtree. This new subtree is generated in an identical fashion to initial trees (see "Initialization" above). A small variation to the standard subtree mutation was also added. With a given probability, the roles of the subtree and the original tree are swapped, i.e., a random node on the randomly generated subtree is replaced with the entire original tree and this becomes the new tree of the offspring.

Add-Equation Mutation. An offspring is subject to "add-equation" mutations, as well as those mutations described above. This mutation produces a new equation tree of depth 1 or 2 using the "ramped half-and-half" method described previously. A new variable v_j, $j > N$ is added to $\mathbf{v}$. This is the variable that the new equation tree will be modifying. The new variable v_j is also randomly incorporated into a randomly selected existing equation, either through a point mutation or a subtree mutation (with equal probability), as described above. Finally, $v_j^{t=0}$ is set to a random value.

Initial-Value Mutation. An offspring's initial values $\mathbf{v}^{t=0}$ are subject to mutation as well. If an initial value is to be mutated, it will either be set to a new random value or perturbed by a value drawn from a given distribution. These two types of initial value mutations occur with equal probability.

Equation Reduction. When an offspring genome is produced, it is checked for possible equation simplifications with a probability of 0.1. For example, the computation $0+1$ will be reduced to 1. Results in (Grouchy and D'Eleuterio, 2010) demonstrated that this improves both solution quality and execution times.

If an extra state variable is not referenced anywhere in the offspring genome (excluding the variable's own equation tree), that variable and its corresponding equation tree are discarded.

Execution

An EMM-based agent's behavior over the course of its lifetime is determined as follows:

1. Set $t = 0$
2. Set $v_i = v_i^{t=0}$, $i = 1, ..., N, ..., N + N'$
3. Update $\mathbf{u}^t$ with current agent inputs (i.e., current sensor values)
4. Evaluate $\mathbf{f}\left(\mathbf{u}^t, \mathbf{v}^t\right)$
5. Update agent output and extra state variables $\mathbf{v}^{t+\Delta t} = \mathbf{f}\left(\mathbf{u}^t, \mathbf{v}^t\right)$
6. Run agent for Δt timesteps using agent output values $v_i^{t+\Delta t}$, $i = 1, ..., N$
7. Set $t = t + \Delta t$
8. Go to step 3 (unless the end of the agent's lifespan has been reached)

The above steps apply to agents operating in a simulation environment as well as to embodied robotic agents operating in the real world. Steps 3-6 are equivalent to propagating agent inputs through an ANN to produce agent outputs for a single timestep of an agent's lifetime in an ANN-based experiment.

Double-T Maze Experiments

In its simplest form, a T Maze test consists of a series of trials where a robot starts in the home position, choses one of the two "arms" of the maze to visit and collects the reward at the end of that arm. In some cases, the robot is automatically returned home once the end of the maze is reached, whereas in others part of the task is for the agents to find their own way home. For each trial, one arm of the maze contains a high reward, while the other contains a low one. The purpose of this task is to demonstrate learning. A successful agent should search both arms for the high reward, and then return to the high-reward arm of the maze in subsequent trials. If the reward is moved, the successful agent should search for and relearn its new position. A Double-T Maze has four arms instead of two, while still only having one high reward (Figure 2). In a discrete maze environment,

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

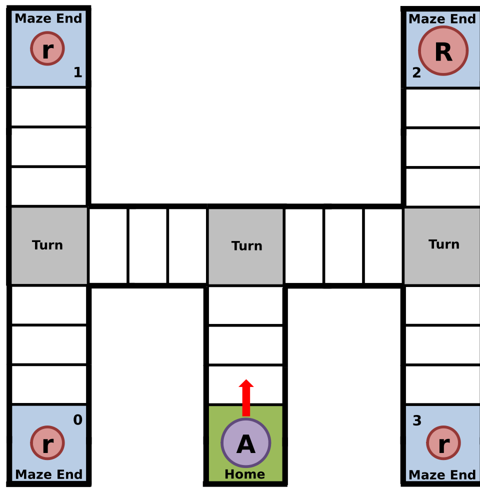

Figure 2: A discrete Double-T Maze. "A" is the agent, "r" are low rewards and "R" is the high reward.

agents must decided to move straight for one unit, or turn left or right, requiring only a single output variable.

The specific T Maze chosen for these experiments was the discrete Double-T Maze with Homing requirements used in (Soltoggio et al., 2008), although certain experimental settings may vary slightly owing to a lack of details in the original paper and a lack of available source code. This version was selected for several reasons. This problem domain has only been solved with plastic ANNs; fixed-weight ANNs have so far been unsuccessful. Furthermore, the Double-T Maze is very difficult, requiring agents to choose repeatedly one of four different movement patterns depending on where they find the high reward. If agents have yet to discover the location of the high reward, they must search up to four different maze arms (requiring four different movement patterns) sequentially. Adding to the difficulty is the requirement that agents must return home after reaching an end of the maze. This doubles the size of each of the four movement patterns. For example, to get to the top left part of the maze an agent must turn left, then right. To then return home, the agent must turn left, then right again. There is a unique four-turn pattern for each of the four arms (LL-RR, LR-LR, RL-RL, RR-LL).

The main point of this experiment is to demonstrate that EMMs can solve this challenging task. However, we attempt to tackle this problem using the same number of fitness evaluations as in (Soltoggio et al., 2008).

Problem Definition. Agent fitness is evaluated over a series of trials. For each trial, the agent is evaluated for a maximum of 35 steps, with each step consisting of one evaluation of the agent's equations and the execution of one move (forward or turn) based on the agent's output v_1. A trial begins with the agent at the "home" position. If an agent executes a turn command while not on a "turn" position (i.e., while on the home position, one of the four reward positions or in a corridor) or executes three consecutive "move forward" commands on a turn position, this is considered a "crash." Crashes end the current trial, returning agents to the home position and subtracting 0.4 from their total fitness. If an agent completes a trial without returning to the home position, a penalty of 0.3 is applied to their total fitness. If the agent reaches one of the three low-reward arms of the maze, a score of 0.2 is added to their fitness. The high reward arm yields a fitness boost of 1.0. When an agent reaches the end of a maze arm, it is automatically turned $180°$. Corridors and turn points last for three forward steps each.

Agents have access to four inputs, "turn," "maze end," "home" and "reward" (u_1, u_2, u_3 and u_4, respectively). The turn input is set to 1.0 when the agent is on a turning point, 0.0 otherwise. The maze-end input is set to 1.0 when the agent is at the end of one of the four maze arms, 0.0 otherwise. The home input is set to 1.0 when the agent is at the home position, 0.0 otherwise. Finally, if the agent collects a low reward, the reward input is set to 0.2 for one step. Collecting a high reward sets the reward input to 1.0 for one step. The reward input is 0.0 at all other times.

Agents have one output v_1. If $v_1 < -0.33$, the agent performs a left turn and then moves forward one unit. If $v_1 > 0.33$, the agent performs a right turn and then moves forward one unit. Otherwise, the agent moves forward one unit in its current direction. All inputs and references to variables v_i, $i = 1, ..., N + N'$ are subject to noise by adding a random value taken from the uniform distribution $[-0.005, 0.005]$ at each equation evaluation.

Agent fitness is evaluated on a set of 200 trials, with the high reward randomly positioned for the first trial. The high reward is randomly repositioned after a randomly selected number of trials H_t, with $35 \leq H_t \leq 65$. Reward repositioning happens three to four times per 200 trial run, with H_t being regenerated after every repositioning. During the evolutionary runs, the first four high reward positions were forced to be distinct. This was not enforced for the 100 sets of 200 trials used for testing top agents.

EMM Algorithm Details. For this task, an island model is used[1]. Each of the 10 islands has a population of 100, giving a total population size of 1,000. Islands are arranged in a ring formation, with the top agent from each island being copied (migrating) to the left or right island every 20 generations. The direction of migration is constant across all islands and switches after each migration. Each island population is tested on its own set of 200 trials, with all 10 sets being regenerated after each generation. Tournament selec-

[1]The source code for these experiments is available for download at `http://www.sr.utias.utoronto.ca/images/downloads/grouchy_alife_2k14.zip`.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

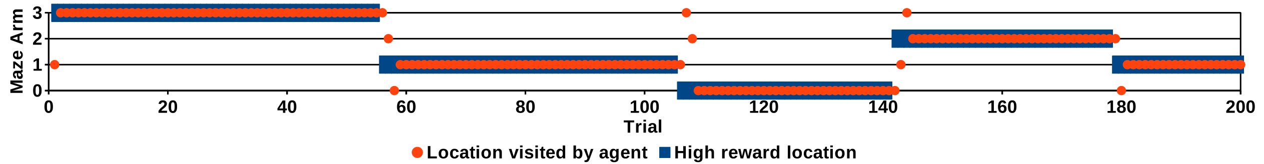

Figure 3: High reward location and maze end visited by a successful EMM agent for each trial of a 200 trial run. Locations 0,1,2,3 are indicated in Figure 2.

tion with a tournament size of 15 is used, and each island's top agent is cloned for the next generation (elitism).

When random constants are needed, they are selected from the uniform distribution $[-5, 5]$. Random initial values $\mathbf{v}^{t=0}$ are set to values from the uniform distribution $[-1, 1]$.

Offspring genomes are produced as a clone of a single parent with a probability of 0.3, otherwise tree-level sexual recombination can occur on one or more of the offspring's $N + N'$ trees with a probability of $0.5/\left(N + N'\right)$ per tree. Tree mutations can happen to any offspring genome with a probability of 0.1 per offspring tree, whereas an extra state variable/equation tree is also added to a genome with a probability of 0.1 per previously existing offspring tree. Offspring are required to undergo at least one tree or add-equation mutation. An initial value is mutated with a probability of $0.1/\left(N + N'\right)$.

If a tree mutation is to occur, it will be a point mutation with a probability of 0.5, otherwise it will be a subtree mutation. If a point mutation is to occur and the tree in question contains at least one constant, a perturbation of a constant mutation will happen with a probability of 0.5, adding a random value taken from a Gaussian distribution with mean 0 and standard deviation 0.5 to a randomly selected constant. During subtree mutation, the original tree and the randomly generated subtree are swapped with a probability of 0.05. Initial values are perturbed using a value selected from a Gaussian distribution with mean 0 and standard deviation 0.25.

Output values v_1 are capped to the range $[-1, 1]$, however extra state variables $v_j, j = 2, ..., N'$ are unbounded. If there is a zero-divided-by-zero operation, or if any variable exceeds the minimum or maximum allowable values of the programming language being used, the current trial is terminated. A maximum genome size of 200 nodes was imposed across all experiments.

At the end of every generation, the top agent from each island is tested on a fixed test set of 100 randomly generated 200 trial runs. Each experiment is run for 1,000 generations, and the final result is taken to be the agent that performs the best on the test set. This is notably different than the experiments in (Soltoggio et al., 2008) where only the top agent in the final generation is tested.

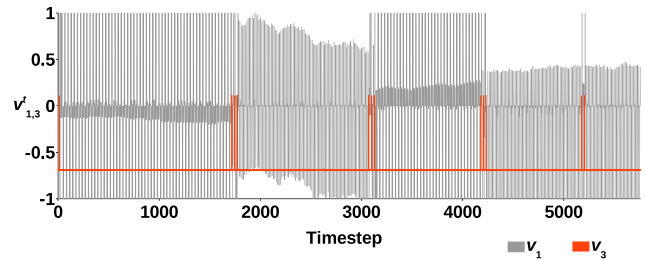

Figure 4: v_1^t and v_3^t values over time for the same EMM agent and 200 trial run as shown in Figure 3.

Algorithm	Test Score		# Successful
	μ	σ	
10 islands w/o extra eqns	95	10	0
10 islands w/ extra eqns	172	18	9
40 islands w/ extra eqns	186	5	18

Table 1: Double-T Maze results from 50 evolutionary runs. A "successful run" has occurred if an agent scores 189.4 or higher on the test set. μ is the mean and σ is the standard deviation.

Results

For the test set used in these experiments, we calculated the "worst-case perfect test score" to be 189.4 (the theoretical maximum test score was calculated to be 197.24). This value is the average score of a perfect agent across the 100 test runs, assuming the agent *always* searches each low-reward arm once before finding the high reward (hence "worst-case") and always returns to the high-reward arm once it is discovered (hence "perfect agent"). Thus we consider an agent with a test score of 189.4 or higher to be a solution to this Double-T Maze. Table 1 shows the results from 50 runs using the same test set, but with different initial populations and different training sets. The runs with 10 islands use the same number of fitness evaluations as in (Soltoggio et al., 2008), although with significantly more agents tested. Results with 10 islands and extra equations/state variables disabled are shown, demonstrating significantly worse per-

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

formance and an inability to fully solve the task. Experiments with 40 islands are also reported, demonstrating performance improvements given more fitness and test evaluations.

From these results, we can conclude that EMMs can successfully solve this difficult task requiring learning. A successful agent with a test score of 191.884 will be examined further. Figure 3 shows how this agent performs on a single 200-trial run. The location of the high reward is shown for each trial, as well as the maze arm visited by the agent. The agent's foraging pattern seems to be fixed: arm 1, arm 3, arm 2, arm 0, then back to arm 1 and the pattern repeats.

The full system of equations (with rounding and simplifications) of this top agent is

$$v_1^{t+\Delta t} = u_1^t v_2^t v_4^t \quad (8)$$
$$v_2^{t+\Delta t} = 5.92 v_3^t / v_2^t \quad (9)$$
$$v_3^{t+\Delta t} = u_2^t - u_4^t - 0.69 \quad (10)$$
$$v_4^{t+\Delta t} = -0.41 u_1^t v_7^t \quad (11)$$
$$v_5^{t+\Delta t} = -0.33 v_6^t \quad (12)$$
$$v_6^{t+\Delta t} = 0.65 - v_8^t - (1.18/v_3^t) \quad (13)$$
$$v_7^{t+\Delta t} = v_5^t - 0.48 \quad (14)$$
$$v_8^{t+\Delta t} = v_9^t(0.02v_9^t + 0.05v_{10}^t - 0.23) - v_5^t - 0.14v_{10}^t + 0.48 \quad (15)$$
$$v_9^{t+\Delta t} = 56.11 u_2^t \quad (16)$$
$$v_{10}^{t+\Delta t} = 0.80 - v_4^t \quad (17)$$

Note that v_1 is the agent's output variable and evolved initial conditions $\mathbf{v}^{t=0}$ are omitted. Figure 5 shows the relationships between variables within this agent's evolved equations. The only equation containing the variable u_4 (the "reward" input) is (10), and its corresponding state variable v_3 seems to be modulating learning. The value of v_3^t is constant at -0.69, except in the cases where an agent is at a maze end ($u_2^t = 1$) and a low reward is collected ($u_4^t = 0.2$). In these cases, $v_3^{t+\Delta t} = 1 - 0.2 - 0.69 = 0.11$. Figure 4 shows the agent's behaviors (i.e., its output values v_1) and the values of v_3 over all timesteps from the same 200 trial run shown in Figure 3. One can clearly see the neuromodulation-like behavior of v_3, as the agent output patterns (v_1) do not change when $v_3^t = -0.69$, however these patterns *do* change when $v_3^t = 0.11$. A positive spike of v_3 seems to cause the agent to try the next arm in its forage pattern (learning), while a fixed negative v_3 value causes the agent to revisit the same arm (no learning). Thus neuromodulation-like behavior has evolved without special neural structures having been specified *a priori*.

Conclusions

We have presented a novel Artificial Life paradigm that uses Evolvable Mathematical Models (EMMs) as controllers for autonomous agents. A Genetic Programming algorithm

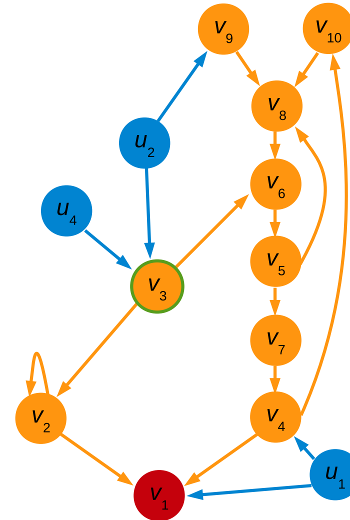

Figure 5: Relationship between variables within the successful EMM agent's evolved equations. Inputs are shown in blue, the agent's output v_1 is red and the extra state variables are orange. Data require one timestep to traverse an orange arrow, whereas input data traverse blue arrows instantaneously. The state variable v_3 is emphasized in green as it plays the role of neuromodulator. The calculations performed at each node are shown in (8) to (17).

was used to evolve systems of equations that map agent inputs to outputs. Functions are represented in the genome as variable-length trees, one for each agent output, and are composed of the four basic mathematical operators, addition, subtraction, multiplication and division, as well as input and output variables and constants. These EMMs can approximate any analytic function. Further trees and corresponding extra state variables can be added through an "add-equation" mutation. Experiments were performed on the challenging Double-T Maze with Homing domain, a task previously only solved using artificial neural networks with connection weight plasticity. Solutions to this domain were evolved successfully using fixed-structure EMMs and the same number of fitness function evaluations as a previous ANN-based experiment. These solutions demonstrated neuromodulation-like learning behaviors without any special neuroplasticity or neuromodulation structures having been specified *a priori*. Furthermore, evolved solutions are readily examinable, as they are represented directly as mathematical equations and are thus amenable to mathematical analysis.

The work presented here is intended as a first step towards evolving autonomous agents as mathematical models of behavior. Future work should look to improve the evolvability of EMMs through alternative genetic encodings (e.g., Linear GP) and different types of evolutionary search. The

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023

results presented here demonstrate the ability of EMMs to evolve behaviors similar to those produced by neuromodulated plastic neural networks. A question that then arises is whether EMMs can be evolved to model behaviors produced by other types of neural networks, such as biological neural networks. Further experimentation on even more challenging domains are required to explore the effectiveness of Evolvable Mathematical Models as autonomous agent controllers.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

Adorni, G., Cagnoni, S., and Mordonini, M. (1999). Genetic programming of a goal-keeper control strategy for the robocup middle size competition. In *Genetic Programming*, pages 109–119. Springer.

Andre, D. and Teller, A. (1999). Evolving team darwin united. In *RoboCup-98: Robot soccer world cup II*, pages 346–351. Springer.

Bailey, C. H., Giustetto, M., Huang, Y.-Y., Hawkins, R. D., and Kandel, E. R. (2000). Is heterosynaptic modulation essential for stabilizing hebbian plasiticity and memory. *Nature Reviews Neuroscience*, 1(1):11–20.

Burbidge, R., Walker, J. H., and Wilson, M. S. (2009). Grammatical evolution of a robot controller. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 357–362. IEEE.

Burbidge, R. and Wilson, M. S. (2014). Vector-valued function estimation by grammatical evolution for autonomous robot control. *Information Sciences*, 258:182–199.

Dürr, P., Mattiussi, C., and Floreano, D. (2006). Neuroevolution with analog genetic encoding. In *Parallel Problem Solving from Nature-PPSN iX*, pages 671–680. Springer.

Durr, P., Mattiussi, C., Soltoggio, A., and Floreano, D. (2008). Evolvability of Neuromodulated Learning for Robots. In *The 2008 ECSIS Symposium on Learning and Adaptive Behavior in Robotic Systems*, pages 41–46, Los Alamitos, CA. IEEE Computer Society.

Floreano, D., Durr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62.

Floreano, D. and Mondada, F. (1996). Evolution of Plastic Neurocontrollers for Situated Agents. In *4th International Conference on Simulation of Adaptive Behavior (SAB'1996)*. MA: MIT Press.

Grouchy, P. and D'Eleuterio, G. M. T. (2010). Supplanting neural networks with ODEs in evolutionary robotics. In *Simulated Evolution and Learning*, pages 299–308. Springer.

Grouchy, P. and Lipson, H. (2012). Evolution of self-replicating cube conglomerations in a simulated 3D environment. In *Artificial Life*, volume 13, pages 59–66.

Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. Wiley, New York.

Jesper, B. and Floreano, D. (2002). Levels of dynamics and adaptive behavior in evolutionary neural controllers. In *From Animals to Animats 7: Proceedings of the Seventh International Conference on Simulation of Adaptive Behavior*, volume 7, page 272. MIT Press.

Koza, J. R. (1992). Genetic programming: on the programming of computers by means of natural selection (complex adaptive systems).

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.

Koza, J. R. and Rice, J. P. (1992). Automatic programming of robots using genetic programming. In *AAAI*, volume 92, pages 194–207.

Mattiussi, C. and Floreano, D. (2004). Evolution of analog networks using local string alignment on highly reorganizable genomes. In *Evolvable Hardware, 2004. Proceedings. 2004 NASA/DoD Conference on*, pages 30–37. IEEE.

Niv, Y., Joel, D., Meilijson, I., and Ruppin, E. (2002). Evolution of reinforcement learning in uncertain environments: A simple explanation for complex foraging behaviors. *Adaptive Behavior*, 10(1):5–24.

Nordin, P. and Banzhaf, W. (1995). Genetic programming controlling a miniature robot. In *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67. Citeseer.

Nordin, P. and Banzhaf, W. (1997). An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior*, 5(2):107–140.

Pascual-Leone, A., Amedi, A., Fregni, F., and Merabet, L. B. (2005). The plastic human brain cortex. *Annu. Rev. Neurosci.*, 28:377–401.

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Soltoggio, A., Bullinaria, J. A., Mattiussi, C., Dürr, P., and Floreano, D. (2008). Evolutionary advantages of neuromodulated plasticity in dynamic, reward-based scenarios. In *ALIFE*, pages 569–576.

Soltoggio, A., Durr, P., Mattiussi, C., and Floreano, D. (2007). Evolving Neuromodulatory Topologies for Reinforcement Learning-like Problems. In *IEEE Congress on Evolutionary Computation (CEC 2007) 25-28 Sept. 2007*, pages 2471–2478. IEEE Press.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2003). Evolving adaptive neural networks with and without adaptive synapses. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 4, pages 2557–2564. IEEE.

Tuci, E. and Quinn, M. (2003). Behavioural plasticity in autonomous agents: a comparison between two types of controller. In *Applications of Evolutionary Computing*, pages 661–672. Springer.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/681/1901537/978-0-262-32621-6-ch108.pdf by guest on 09 February 2023