Evolving Hebbian Neural Networks for Articulated Robot Locomotion

Christian Hahm
christian.hahm@temple.edu

Department of Computer and Information Sciences
Temple University

January 9, 2024

Abstract

A program called the *Animat Creator* was developed that allows a user to evolve Hebbian neural networks ("brains") for the task of articulated robot locomotion. Rather than focusing on any one single neuroevolution algorithm, the program was implemented to flexibly allow the "plug-and-play" of any arbitrary neuroevolution algorithm. As such, various neuroevolution methods were implemented and experimentally tested on simulated segmented hexapod and quadruped robots. The algorithms successfully produced brains that control the articulated robots to walk. In addition to the *Animat Creator*, associated tools called the *Genome Editor* and the *Brain Viewer* were created that allow visualizing brain genomes and brains respectively.

Keywords: *Hebbian network, mobile robot, articulated robot, artificial animal, neuroevolution, modularity, regularity, genetic encoding, genetic algorithm, simulation, artificial life, artificial intelligence*

Contents

1 Introduction

In preparation to set up an ecosystem of artificial animals (aka "animats", as coined by [Wilson, 1986, p.2]), the first requirement is to create a bare-minimum animat species with which to seed the world. "Bare-minimum animat" means we would expect any given animat species to be autonomously capable of at least 3 non-trivial behaviors:

- *locomotion* (to move its sensors and motors around the environment)
- *food-seeking* (to sustain itself as it burns energy)
- *reproduction* (to perpetuate and guide evolution)

Note that locomotion is a pre-requisite for the food-seeking behavior, and also for the reproduction behavior in the case of selective sexual reproduction, since those require close proximity. So, evolving locomotion is fundamental. Towards this end, a program called the *Animat Creator* was created, which gives users the ability to design and evolve the brains of articulated robots to locomote.

Within the *Animat Creator*, various methods of neuro-evolution were implemented and experimentally tested to compare their results with respect to evolving robot locomotion. In order to test multiple neuro-evolution frameworks, the *Animat Creator* uses a general "plug-and-play" mechanism which allows the developer to easily plug in any arbitrary neuro-evolution algorithm/encoding of their choice, so long as the end product brain conforms to a certain interface.

Multi-leg robot locomotion is a classic problem in neuroevolution. The goal of the task is to evolve a neural network that controls a robot with legs so that the robot walks. The problem has been tackled with various methods. It is a difficult problem because the brain must coordinate the control of multiple legs.

A major consideration when approaching the problem is to use a genetic encoding which allows/encourages *modularity* and *regularity*. Both concepts are concerned with compressed representations of the neural network. *Modularity*, in the context of multi-legged locomotion, means that the genetic algorithm does not need to separately evolve the subnetwork of each individual leg. Instead, the algorithm only needs to evolve one subnetwork to control a single leg; then, it can simply instantiate the subnetwork, as a module, for any number of legs [Gruau, 1994, p.3]. *Regularity* means that symmetries and repetitions in the neural network can be expressed using a more compact representation than explicitly describing the entire network [Clune et al., 2011].

2 Related Works

Direct encodings of neural networks represent/evolve neural networks neuron-by-neuron and connection-by-connection. Therefore, they are not well suited for regularity and modularity, since every part of the network is explicitly and independently represented in the genome data structure. *Indirect encodings*, on the other hand, do not explicitly represent the neural network, but rather provide information by which to generate the neural network. For example, a neural network may be generated by a computer program, as in a sequence of instructions. If a group of those instructions are re-used multiple times in generating the network, it is like a *function* or *module*; since the same module is used in generating different parts of the network, those parts may have similarities in their structure or function.

[Sims, 1994] is the classic work on evolving simulated 3D articulated robots, evolving *both* robotic morphology (i.e., body) and neural network within a single genome. Remarkable results were achieved due to the flexible (indirect) genetic encoding, which in this case used a graph/network data structure. Neural networks for each body segment were stored in each node of the graph. Interestingly, the neurons used were not traditional neurons (i.e., summing their inputs then performing a threshold or activation function); instead, each neuron computes a mathematical function (e.g., absolute value, sine, sum), thus the number of inputs to the neuron depends on the function, and cannot have arbitrarily evolved connectivity. The resulting "virtual creatures" exhibited naturalistic locomotion behaviors like swimming, flying, and walking. The genetic encoding for the body is used in *Section 3.1.2*.

[Gruau, 1995] used an indirect genetic encoding called Cellular Encoding (CE) to evolve a neural network (both its static weights and topology) that controls the locomotion of a hexapod robot. The key advancement

in CE compared to traditional neural encodings was to represent the neural network *indirectly*, using genetic programs, rather than directly. CE is a *developmental encoding* in the sense that the network incrementally develops over time starting from a single ancestor cell, according to the evolved cellular program (genome). In CE, the genome is represented using a sequence of trees. The trees contain "cellular instructions" that are executed sequentially, like a computer program, to generate the neural network. For the hexapod locomotion problem, [Gruau, 1995, p.8] described hand-designing the modules within an initial genome, such that part of the genome controls a single foot, part of the genome controls a single leg, part controls half of the body, etc. This same method is used in *Section 4.1*.

A critical review of CE [Hussain, 1997, p.46] concluded by recommending to test an "axonal growth" extension to CE by [Kodjabachian and Meyer, 1998] called Simple Geometry-Oriented Variation of Cellular Encoding (SGOCE). SGOCE is like CE, except the developmental cells exist in a 2D discrete space/substrate where they can move, and interact. The use of spatial representation allows for the cells to connect freely using a "*GROW* axon" instruction, which is elegant and facilitates arbitrary connectivity. Furthermore, this gives the opportunity for neurons to organize into topographic layouts. [Kodjabachian and Meyer, 1998] used the SGOCE paradigm to evolve a neural network (both its static weights and topology) that controls the locomotion of a hexapod robot.

Direct encodings usually only evolve the weights of a fixed-topology neural network. NeuroEvolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen, 2002] is a direct encoding that improves on prior direct encodings by evolving the topology of the neural network (i.e., the nodes and connectivity) in addition to the connection weights. The neural networks are encoded elegantly and simply using two linear genomes: one for neurons, one for connections.

HyperNEAT [Stanley et al., 2009] is an indirect encoding, built on NEAT, that takes an approach intermediate between direct encodings and developmental encodings. A HyperNEAT genome is represented using a compositional pattern producing network (CPPN), which is a network composed of simple geometric patterns that combine to output a complex pattern. This complex pattern is akin to a morphological or spatial pattern one might find after multicellular biological development, though bypassing the arduous developmental process. In HyperNEAT, the CPPN is queried to generate the connectivity of a given neural network substrate. Due to its geometric nature, the connectivities of neural networks generated by HyperNEAT tend to exhibit regularities (e.g., symmetries, repetitions, etc.) [Clune et al., 2011] but not usually modularity [Clune et al., 2010]. It is possible to encourage modularity by adding an extra Boolean output to the CPPN called link-expression output (LEO) [Verbancsics and Stanley, 2011]. Furthermore, the substrate itself can be evolved using the Evolvable-Substrate HyperNEAT (ES-HyperNEAT) extension [Risi and Stanley, 2012].

[Clune et al., 2009] used HyperNEAT to evolve the static weights of a neural network that controls the locomotion a simulated 3D quadruped robot. The neural network was 3-layer feedforward. The quadruped robot was shaped like a table, with a single square segment for the torso and 4 articulated legs (with 3 hinge-joints each). Each leg had a Boolean touch sensor indicating whether or not the leg is touching the ground. The 20 inputs to the neural network were the current joint angles of the legs, the legs' touch sensors, the rotational angles of the torso, and finally a sine function. The 12 outputs of the neural network controlled the leg joints. HyperNEAT was able to evolve a network which could locomote the robot, though it required the explicit inputting of a periodic (sine) function to the neural network so that it could rythmically control the motor.

Recently, [Najarro and Risi, 2020] evolved Hebbian (and static) neural networks to control the locomotion of simulated 3D quadruped robots. The neural network was 3-layer feedforward, using *tanh*-activated continuous neurons. The robot was shaped like a 4-legged spider, with a single spherical torso and 4 articulated legs (with 2 hinge-joints each). The 28 inputs to the neural network were the positions and velocities of body segments. The 8 outputs of the neural network controlled 8 leg joints. The neural network operated using the ABCD Hebbian plastic mechanism, used in *Section 3.1.1*. It was found that, when the networks encountered situations for which they were trained, the static-weighted networks slightly outperformed the Hebbian networks. However, when the networks encountered situations for which they were not trained, for example when one of the leg's morphology was damaged, no static-weighted network was able to cope with the damage at all. On the other hand, it was found that in the Hebbian networks, unlike static networks, some robots adapted their gait to overcome their leg damage and still successfully complete the locomotion task. This suggests that the ABCD Hebbian model, which is a parallelizable real-time learning algorithm, allows for robust and adaptive behavior in robots, and warrants further exploration.

3 Framework

3.1 Animat

An animat consists of two high-level components: a *brain* and a *body*. Their frameworks can be entirely separate from each other. The only technical requirement is that the two frameworks interface with each other to communicate information.

More technically, the body and brain interface at the point of sensory-motor neurons. When a physical event occurs on/in the body, the body sends that information to the brain by activating a special neuron called *sensory neuron* (aka input neuron). The brain, conversely, can send information to the body by activating a special neuron called *motor neuron* (aka output neuron). Non-sensory-motor neurons are called *interneurons* (aka hidden neurons).

3.1.1 Brain (Hebbian network)

The *continuous* model was used for the neuron, and a *Hebbian learning* model was used for the connections. The algorithm operating the brain was entirely parallelized. As such, a brain of any size operates in the same amount of constant time $O(1)$. An animat's brain is updated once per *update period* $T = 0.1$ seconds. Brains were permitted to evolve recurrent connections.

Neurons Here, a neuron outputs a continuous value. The range of values a neuron can output is determined by its *activation function*, which here was made evolvable. The available activation functions were:

- Sigmoid
$$y = \frac{1}{1 + e^{-\alpha x}} \qquad y \in (0, 1)$$
- Tanh
$$y = tanh(\alpha x) \qquad y \in (-1, 1)$$
- Leaky ReLU
$$y = \begin{cases} x, & x >= 0 \\ \alpha x, & x < 0 \end{cases} \qquad y \in (-\infty, \infty)$$

The slope of each activation function is modulated by an evolvable parameter $\alpha \in \mathbb{R}_{\geq 0}$. A larger α results in a steeper slope (i.e., the neuron activates more easily, with weaker inputs), whereas a smaller α results in a more gradual slope (i.e., the neuron is harder to activate, requiring stronger inputs).

Activation functions can vary in the hidden neurons throughout the network, depending on how they are specified in the brain genome. However, motor neurons were specially restricted to the *tanh* activation function to constrain their output range to $(-1, 1)$, such that -1 represents activating the motor to its most extreme negative angle, whereas 1 represents activating the motor to its most extreme positive angle.

Each neuron has two evolvable parameters in addition to the activation function. First is a *bias* $b \in \mathbb{R}$, which is the "base" input to the neuron. Second is a *sign*, a Boolean flag which, if set, flips the sign of the neuron's output (i.e., from positive to negative, or from negative to positive).

Hebbian Learning Natural brains (neural networks) learn by a process called "Hebbian learning" [Hebb, 1949]. Though we cannot fully simulate the wetware of an animal to recreate "true" Hebbian learning, there are mathematical abstractions of the Hebbian learning process. Here, the ABCD Hebbian model was used, as in [Najarro and Risi, 2020], due to the results reported by [Najarro and Risi, 2020] of adaptive behavior observed in an evolved quadruped robot.

In the ABCD model, each weight w on a connection c increments its magnitude by Δw at time t according to the following equation [Najarro and Risi, 2020, p.4]:

$$\Delta w = r * (A * a_{pre} * a_{post} + B * a_{pre} + C * a_{post} + D) \tag{1}$$

Where $\{A, B, C, D\}$ are the evolved Hebbian learning coefficients for that specific connection, r is the evolved learning rate, a_{pre} is the activation of c's pre-synaptic neuron at time t, and a_{post} is the activation of c's post-synaptic neuron at time t.

3.1.2 Body (articulated robot)

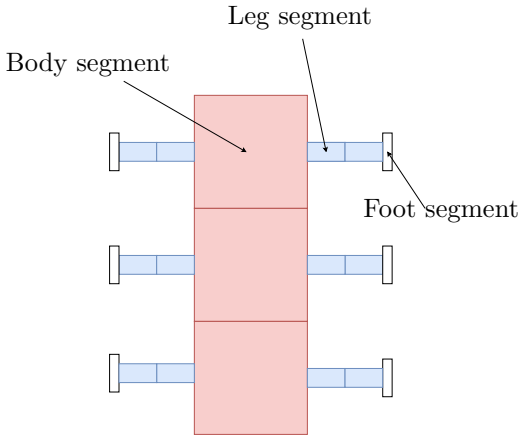

Figure 1: A diagram of a hexapod robot used in the experiments.

Morphology [Sims, 1994] details a genetic encoding for evolving the morphology of 3D articulated robots. The encoding is human-readable: one can use it hand-design a robot morphology. For example, one could easily design genomes for insectoids, humanoids, and trees, as in [Sims, 1994, p.4]. In a Sims genome, each node of the graph represents a body segment type, and each edge in the graph represents a physical connection between body segments.

In the experiments, segmented bug robots similar to Sims' were used (see *Figure 1* for a diagram). Specifically, variously-sized hexapod (3 torso segments, 6 legs) and quadruped (2 torso segments, 4 legs) were tested. The robot contained 7 total body segments per torso segment. The body segments were cuboids hierarchically connected by 3D (aka ball-and-socket, spherical) joints. There were large body segments comprising a column, the "torso" or the agent's central body. Each torso segment had a pair of legs, one leg on the left and the other on the right. A leg itself consisted of three total body segments: an upper leg segment (thigh), a lower leg segment (calf), and a foot. The hexapod robot thus contained 21 total body segments, whereas the quadruped contained 14 total body segments.

Sensors For the task of locomoting in a fixed global direction, the body was equipped with forms of *proprioception* (i.e., a sense of body orientation) and *somatosensation* (i.e., a sense of physical touch). For proprioception, for each body segment, the values of a quaternion $Q =< w, x, y, z >$ representing the body segment's global rotation were used as the sensory neuron activations, where $w, x, y, z \in [-1, 1]$. For somatosensation, each of the 6 faces of cuboid body segments were equipped with a touch sensor. A touch sensor outputs 1 when colliding with an object, such as wall or ground; it outputs 0 otherwise. Sensor values were input to the neural network at each timestep. In total, each body segment contained 10 sensory neurons (6 touch sensors, 4 rotation sensors). Thus, a hexapod robot (with 21 segments) contained 210 sensory neurons, whereas a quadruped robot (with 14 segments) contained 140 sensory neurons.

Motors The body was controlled by motor neurons which drive the joint angle between segments. Each body segment had 3 associated motor neurons, which control the joint's $X-$,$Y-$, and $Z-$drives. Thus, a hexapod robot (with 21 segments) contained $21 * 3 = 63$ motor neurons, whereas a quadruped robot (with 14 segments) contained $14 * 3 = 42$ motor neurons. The drive angles of the robot were limited to $[-45, 45]$ degrees. Segment mass in addition to joint stiffness and damping were tweaked throughout the experiment, for testing and due to issues with stability in the physics engine. In the end, the values were: `mass` $= 1.25$

per unit volume, `stiffness` = 500, and `damping` = 25.75. Motor values were read from the neural network at each timestep and used to drive the joint motor.

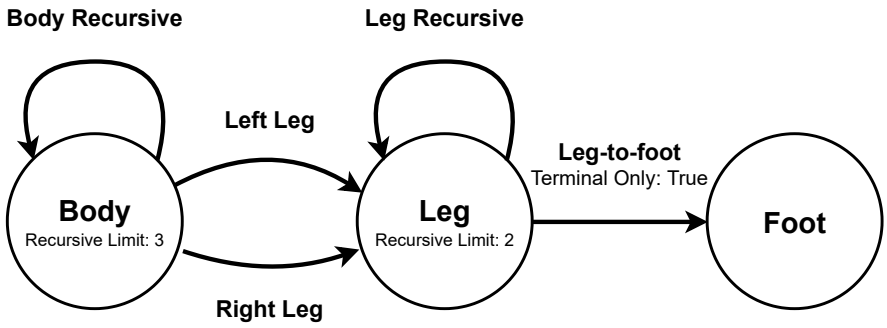

Figure 2: The genome encoding the segmented hexapod robot's morphology.

Genome With 3 types of body parts, the genome required 3 nodes: a *Body* node, a *Leg* node, and a *Foot* node. The *Body* and *Leg* nodes were equipped with recursive connections, so their length could be scaled to any desired value; since a node indirectly encodes a type of body segment (e.g., a leg), it can be re-used as many times as desired and even connected to itself for a long chain of the body part. For example, by simply modifying the recursive limit in the *Body* node, the number of torso segments is modified, allowing to easily generate both hexapod (recursive limit = 3) and quadruped (recursive limit = 2) robots for testing.

3.2 *Animat Creator*

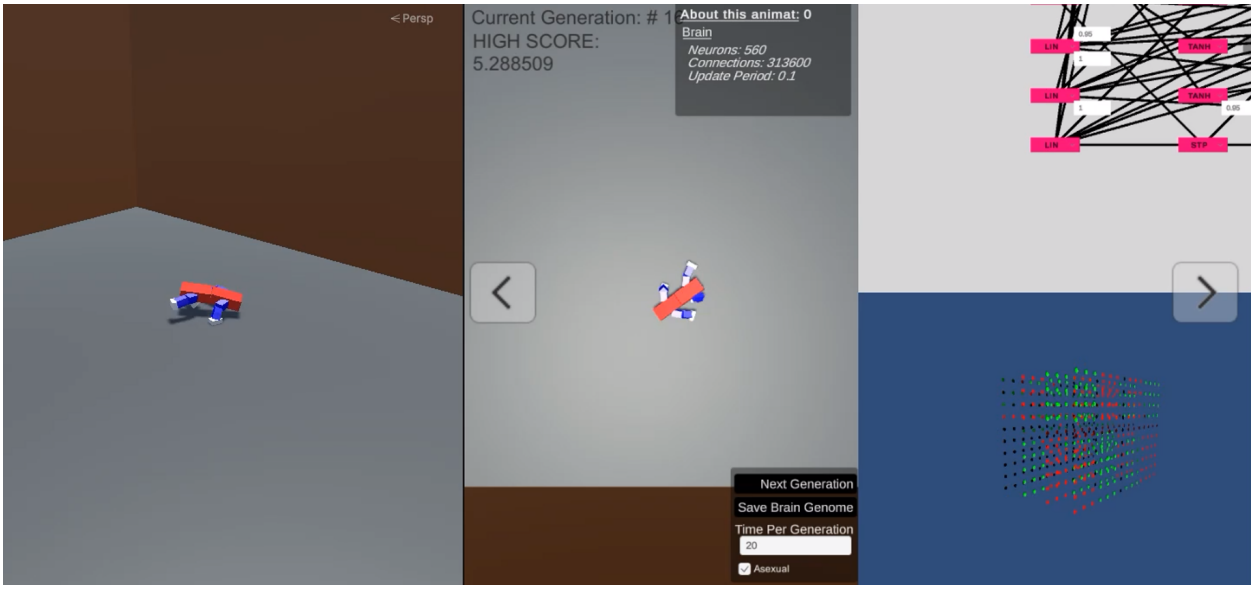

Figure 3: A screenshot from the *Animat Creator* showing a walking quadruped robot.

The *Animat Creator* is an application that was developed for evolving locomoting articulated robots. It contains a 3D scene for evolution and various GUI tools for monitoring the evolutionary process (see *Figure 3*). At initialization, the first generation of animats is spawned. They all have the same initial brain genome,

which is the starting point for evolution. It is possible to manually pre-design genomes so that the process begins at a reasonable starting point, rather than hoping the evolutionary process can develop the whole thing from scratch.

The application gives developers the freedom to plug in any arbitrary genetic (neuroevolution) paradigm for testing. This "plug-and-play" mechanism helps the project avoid limitations that might be imposed by designing the project around any one specific neuroevolution algorithm, and allowed easily switching between paradigms to gather results for comparison. The plug-and-play interface works by associating a unique string label with each body part sensor/motor. The requirement of the genetic encoding is, when generating the neural network from the genome, to associate the string labels of body parts with corresponding sensory-motor neurons. For example, a (sensory) neuron labeled "`LeftBody_MiddleLeg_Foot_RotationSensor_X`" will at any given moment have an activation corresponding to the middle-left foot's x-rotation in space, whereas a (motor) neuron labelled "`LeftBody_MiddleLeg_Foot_Motor_X`" drives the middle-left foot joint's x-rotation.

The *Animat Creator* consists of 3 components:

- The *Incubator*, a 3D scene filled with multiple *training pods* containing individual animats
- The *Genome Editor*, an application that allows viewing and editing brain genomes
- The *Brain Viewer*, an application that is used to view the physical structures and activations of animat brains

An animat's *fitness score* is equivalent to its distance travelled along the positive global z-axis. At the end of each generation (10 to 20 seconds long), the animats were sorted by their fitness scores in preparation for reproduction. The reproduction was 50% asexual and 50% sexual. This means the first half of the offspring genomes were generated by asexually reproducing the top 50% scoring animats. The second half of the offspring genomes were generated by sexually reproducing pairs of animats until the total desired number of offspring was reached; the number of offspring produced in a "litter" by each pair was determined by the proportion of their combined fitness scores to the combined fitness scores of all animats in the generation. After reproduction, all offspring genomes were mutated.

3.2.1 *Incubator*

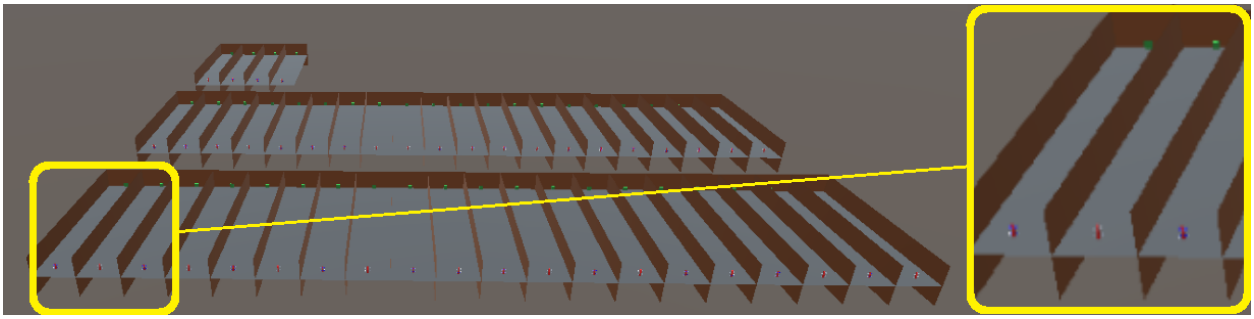

Figure 4: A group of training pods. An animat is at the bottom of each pod.

Evolutionary algorithms require a population of individuals, which represent the latest evolved solutions, and act as the stepping stone to the next generation of solutions. In this case, the evaluation of one individual is independent from any other; therefore, the individuals were kept physically separated, each in their own private container dubbed a *training pod*. A training pod is a long cuboid with 4 walls and a floor (see *Figure 4*). For an experimental trial, the animat is spawned at one end of the training pod, and its fitness is evaluated by how far it is able to travel towards the other end of the pod.

All the training pods and animats were contained in a single physics scene, called the *Incubator*. Since the algorithm to compute collision physics is not constant-time, the number of animats that could be simulated per generation without framerate lag was limited to about 80 or less. As the physics was computed on the

CPU, it also had to compete with the computation of many animat brains (also computed on the CPU). Though there probably is no physics engine with much better efficiency, we could still parallelize the fitness evaluations by running each training pod in its own separate process. Then at the end of all simulations, the results could be aggregated on the single main process. Nonetheless, parallel simulation was not done here as it would have needlessly increased complexity; 50 individuals was sufficient for evolving locomotion.

3.2.2 *Genome Editor*

The *Genome Editor* is an application that was built to allow creating, editing, and viewing animat genomes in a 2D GUI. The way in which the genome is displayed depends on its paradigm. Here, two classes of genomes are made available: *trees* and *graphs*.

Both CE and SGOCE genomes are represented by symbolic instructions in a *tree* data structure (see e.g., *Figures 9,10*). The algorithm to layout the tree GUI was taken from [Mill, 2008]. Each node in a tree genome can be edited. Clicking a node activates a dropdown menu, from which a new cellular instruction may be selected. Since trees can get quite large, left-clicking on a connection will center the screen on a connected node. Right-clicking a connection will insert a new node into the genome at the location clicked.

NEAT, HyperNEAT, and ES-HyperNEAT genomes are represented by a *graph* (aka network) data structure, including nodes and edges (see e.g., *Figure 14*). Each node in a network genome is assigned a *layer*, which determines its x-coordinate. The y-coordinate is determined by the order in which the nodes are drawn, so that nodes drawn in the same layer stack up in a column.

The *Genome Editor* can be used to manually design and edit brain genomes. A manually-created genome can be saved to the disk, then loaded into the *Animat Creator* as an initial genome for testing and evolution. For example, the editor was used to design and test various CE genomes in *Section 4.1*. During the evolutionary process, the *Genome Editor* displays the brain genome of the currently-viewed animat to the user.

3.2.3 *Brain Viewer*

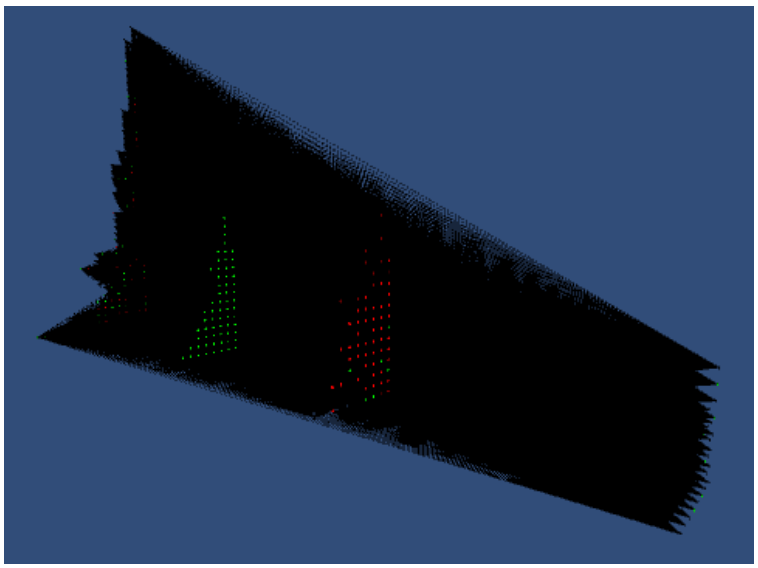

Figure 5: An active brain, visualized in the *Brain Viewer*.

The *Brain Viewer* shows the neural activity and 3D structure of animat brains (see *Figure 5*). A brain's structure, meaning the locations of its neurons and the synapses/connections between its neurons, is generated from a given brain genome and displayed in 3D space. In this way, when a brain genome is being created in the *Genome Editor*, its corresponding brain structure can be viewed in the *Brain Viewer*, which

may be a useful feature for determining how changes to the genome affect changes in the brain structure. Each neuron is displayed as a circle, and each connection is displayed as a line drawn between two neurons.

During the evolutionary process in the *Animat Creator*, the animat's brain and real-time neural activations are displayed in the *Brain Viewer*. This can help the user understand correlations between the animat's brain activations and its behavior. A green neuron represents a positive activation, whereas a red neuron represents a negative activation. The more intense the green/red color of the neuron, the higher the magnitude of its activation (up to a limit of 1, though activation magnitudes can exceed 1 in the case of ReLU); if a neuron is black, that means its activation is zero.

4 Experiments

Figure 6: Quantitative results obtained from each neuroevolution method.

Various neuroevolution methods were implemented and tested, on a population size of about 50. See *Figure 6* for some sample quantitative results from the various methods. From the graphs, it is clear that all methods of neuroevolution were able to evolve improved locomotion results over multiple generations. A qualitative discussion follows.

4.1 Cellular Encoding (CE)

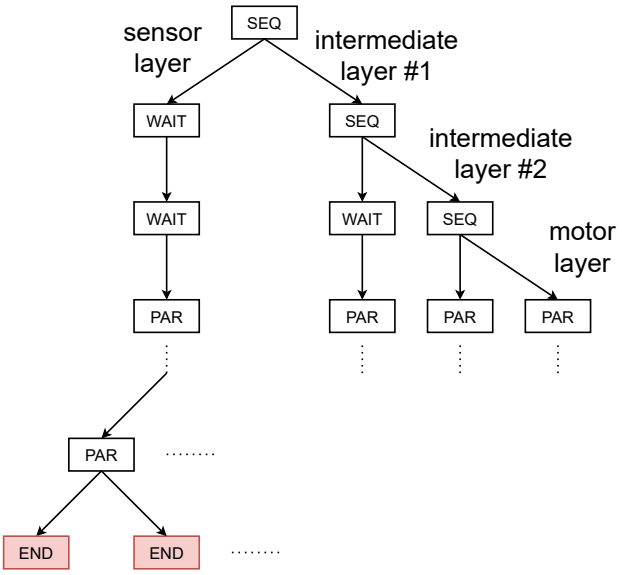

Figure 7: The initial Cellular Encoding single-tree genome.

Cellular Encoding [Gruau, 1994] was tried first. In CE, the genome is a cellular program, represented in the form of a tree. The instructions in the tree tell each cell whether to divide or differentiate at each time step. Each cell has its own pointer to the genetic code, and continues executing the code until the end is reached, after which the neural network is fully formed. In this way, the nodes and connections of the neural network are represented indirectly rather than directly.

There are multiple benefits of this approach compared to direct encodings. One is that this indirect genome, at least in its multi-tree version, is much more compact than directly using a full network representation. Another is that the genomes are somewhat human-readable, since they use understandable cellular instructions. Also, a single mutation in evolution is much more impactful, it can impact multiple substructures throughout the neural network since mutations affect subtrees of instructions followed by multiple cells, instead of mutating singular neurons and connections one at a time. Finally, the phenotype can exhibit regularities since groups of cells follow the same or a similar sequence of cellular instructions.

Theoretically, the CE model supports modularity, though originally this functionality was not fully fleshed out. As noted by [Hussain, 1997, p.20], the first attempts of representing modular genetic code in single-tree CE were implemented using a form of addressing-by-template. By this method, a certain instruction could "call" another subtree like a function using another subtree as a template, so long as the templates of both the caller and the callee trees matched up. It turned out that the template method did not operate very elegantly in trees. As such, [Gruau, 1995] later improved the scheme for modularity by representing genomes in a *sequence of trees*, rather than one single monolithic tree. In this way, an instruction (here called Jump JMP) could be used with relative addressing to "call" another tree in the sequence, like a function in a computer program.

Here, a pared-down version of CE as described in [Gruau, 1994, Gruau, 1995] was implemented. In order to begin evolving a locomoting animat with CE, an initial CE genome had to be hand-designed. Additional cellular instructions were introduced specific to the ABCD Hebbian algorithm, such $INCA$ (increment A), $DECA$ (decrement A), $INCB$, $DECB$, etc. The genome was designed to ensure that all the sensory-motor neurons are properly created and identified. The brain was divided into an input layer (containing the sensory neurons), 1 or 2 hidden layers (containing the hidden neurons), and an output layer (containing the motor neurons).

Mutation operators affected nodes in the tree. During the genome mutation step, every node in the

tree/genome had a chance to be mutated. A mutation hyperparameter m was set so that a genome would undergo approximately $m = 5$ mutations on average. There were 3 types of possible mutations: *add*, *modify*, and *delete*. An *add* mutation adds a random instruction to the tree, as a child to the mutated node. A *modify* mutation changes the mutated node's cellular instruction into a different random instruction. Finally, a *delete* mutation removes the mutated node from the tree. The nodes that were included in the initial genome (i.e., the Jump [JMP] and division [SEQ,PAR] nodes) were protected from mutation, since they were necessary to properly develop the layers and the sensory-motor neurons of the brain.

The initial experimental trials used a single-tree genome (see *Figure 7*). Recombination of two single-tree genomes was achieved by swapping a randomly selected subtree from each (as in Koza's genetic programs [Koza, 1990, p.12-13]). The ancestor cell first performed SEQ (sequential) cellular division into 3 cells: a sensor ancestor cell, a hidden ancestor cell, and a motor ancestor cell. These ancestor cells performed PAR (parallel) cellular divisions until the correct number of cells was reached to align with the sensors and motors. The single-tree did not result in locomotion, or much movement at all. This is likely because my version of the algorithm was bugged, but regardless, the single-tree genome lacked the capacity for modularity as I did not implement addressing-by-template.

In later trials, the single-tree genome was extended into a multi-tree genome, containing a total of 15 trees (see *Figures 8,9*). Recombination of two multi-tree genomes was achieved by two-point linear crossover of the tree sequences. The multi-tree genome facilitated faster evolution than the single-tree genome, since each tree acted as a re-usable module. As similarly designed in [Gruau, 1995, p.8], each tree of the genome was designed to correspond to a hierarchical modular area of the body. For example, Tree #8 represents a module for half of the entire body; Tree #8 would be used/called twice, since there are two halves to the body. Similarly, Tree #9 represents a module for a leg. Since there are 6 legs in a hexapod robot, the leg module would be called 6 times.

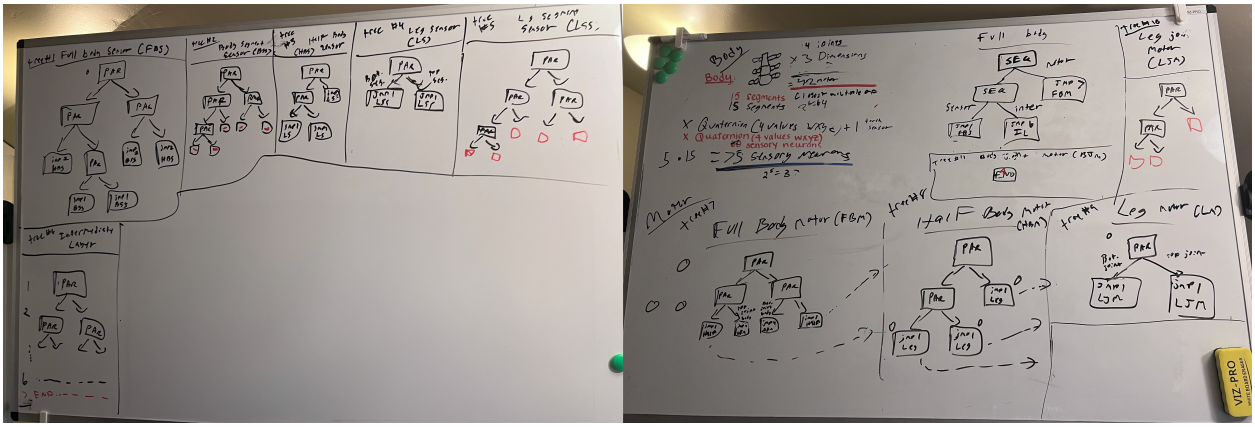

Figure 8: The initial Cellular Encoding multi-tree genome, handwritten with notes and labels.

Overall, Cellular Encoding performed successfully for robot locomotion. The brain in *Figure 5* was developed from a CE genome.[1] The number of neurons in the network was constrained by disabling the mutation of cell divisions. This was to prevent an ambiguous identity (or deletion) of sensory-motor neurons, and also because how best to handle mutations involving two-child cellular instructions (such as cellular division) is unclear. As such, the substrate was pre-specified and could not evolve. Only the neuron *characteristics* were allowed to evolve (e.g., bias, activation function), along with connections between neurons and their characteristics. Sexual reproduction implemented in the form of crossover seemed to perform better (or at least seemed to achieve results faster) than asexual reproduction.

Qualitatively, the best locomotion results achieved in the CE trials can be described as smooth "rhythmic" motion. The evolved hexapod robots tended to shift their limbs on one side of the body forward, then shift them back cyclically, in a regular period. This resulted in the robots shuffling their limbs forward and backward in a rhythmic manner, almost like a gallop. For a video clip, see *Section 6*.

[1] Though note only the z location of neurons is accurate (due to sequential SEQ division); the x and y locations are randomly arranged since there is no spatial axis defined for parallel PAR division

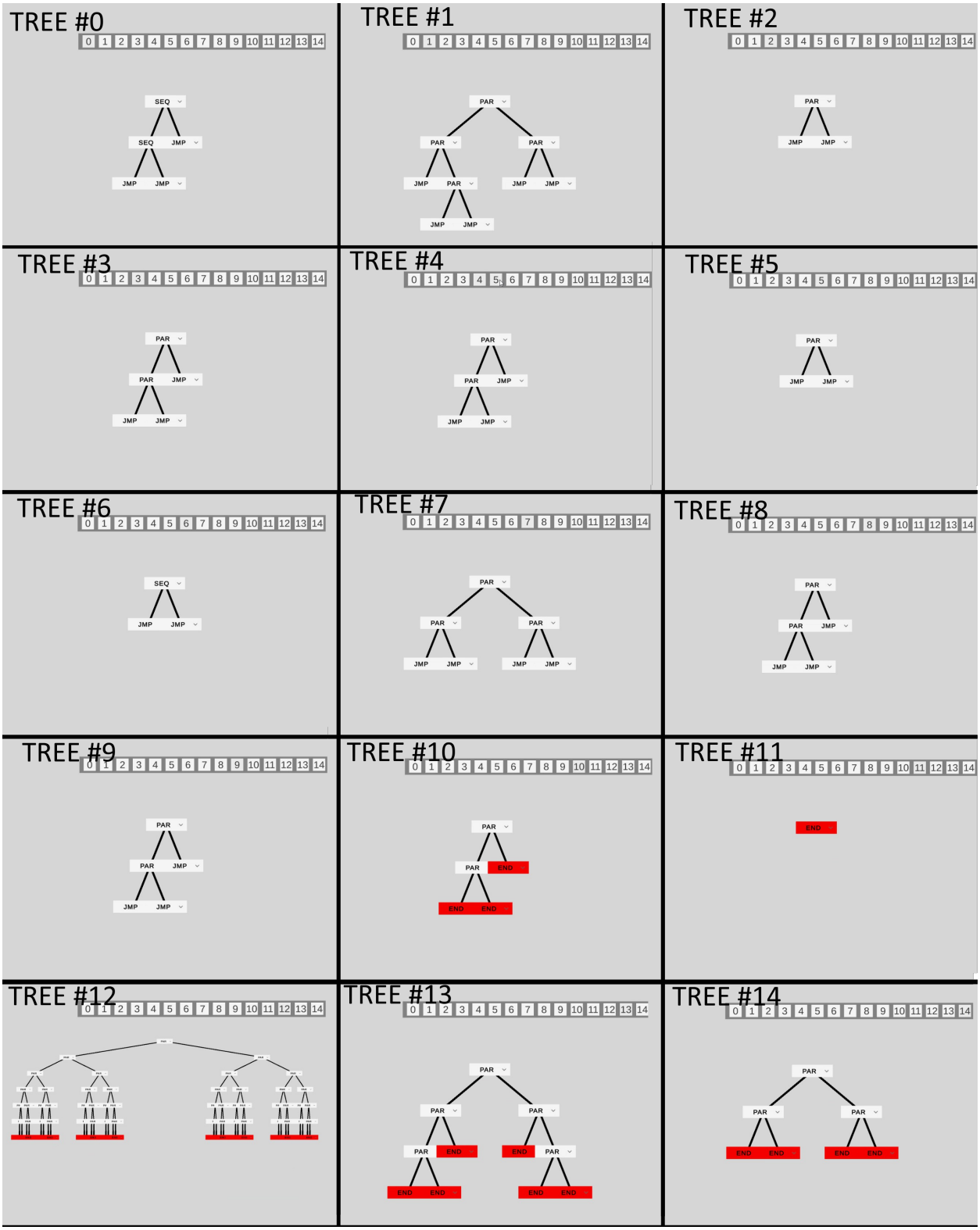

Figure 9: The initial Cellular Encoding multi-tree genome, digitized in the *Genome Editor*.

4.2 SGOCE

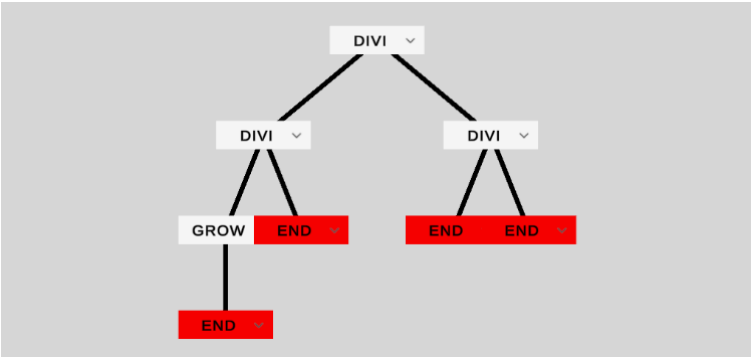

Figure 10: An SGOCE genome in the *Genome Editor*.

While CE seems theoretically neat, a couple of practical issues became apparent during the implementation and testing. Firstly, the instructions were somewhat clunky, as there are 30+ cellular instructions, many of them performing indexed operations on an ever-changing ordered list of synapses. Another major issue, as noted by [Hussain, 1997, p.35], is that while CE theoretically addresses the problem of growing any arbitrary neural network from a single ancestor cell, it is not clear where/how to add the input and output neurons for a given problem, which is often what experimenters wish to do. In the CE trials, to address the problem, the genome was manually hand-designed to ensure that the sensory-motor neurons develop properly and are labeled, though ideally CE would just "automatically" plug them in. Finally, in CE it is not easy for neurons to arbitrarily connect to each other, since they only exist in an abstract data structure rather than physical space. As such, forming new connections requires complicated cellular instructions and referencing, or the inefficient process of first forming excess synapses and then pruning them at a later stage. This issue is easily resolved by the SGOCE extension to CE, which embeds the neurons in physical space and adds new cellular instructions (e.g., grow connection, *GROW*) that allow cells to directly connect to each other.

A version of the SGOCE algorithm was implemented in voxel (3D cellular) space and tested in the *Animat Creator* (see *Figure 11*). As in CE, the process starts with a single ancestor cell, and it follows a cellular program, dividing and differentiating, to create a neural network. Unlike in CE, the cells of SGOCE are embedded in a physical space rather than an abstract data structure. The benefit of representing the cells in physical space is that they have a unique physical location, which means they can locate and interact with each other directly.

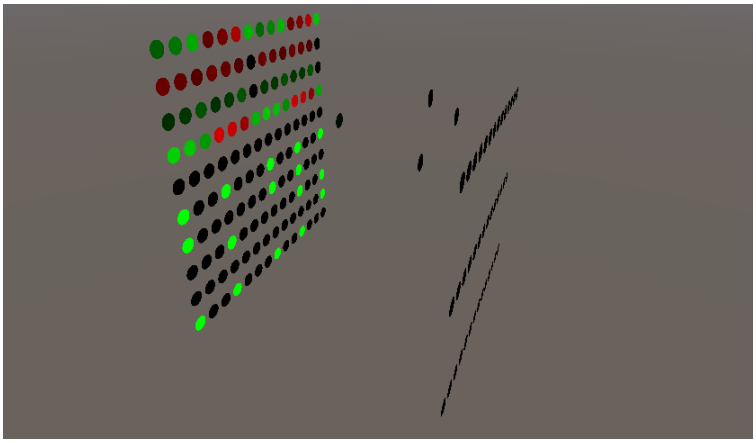

Figure 11: A brain generated by SGOCE. Sensor layer is on the left, motor layer is on the right. The hidden neurons, which grow and divide according to the cellular program, are interspersed throughout the middle.

In testing, the sensory-motor neurons were embedded directly into voxel space before running the CE program, rather than generated from the CE program. Neurons generated by the SGOCE program are only hidden neurons. The hidden cells could immediately begin interacting with the sensory-motor neurons in space by using a *GROW* instruction to create an output synapse in some relative direction according to the vector $\langle x, y, z \rangle$. Similarly, a cell divides in space using a *DIVI* instruction, placing the offspring cell in some relative direction specified by vector $\langle x, y, z \rangle$. The values of the vector are arguments contained in the instruction node. Whenever a node is mutated, either the instruction itself changes or the instruction is left alone and its arguments are mutated. The more arguments an instruction has, the more likely the arguments are to be mutated rather than the instruction. More technically, an instruction with n arguments has a $\frac{1}{n+1}$ chance of the instruction changing, and an $\frac{n}{n+1}$ chance of the arguments changing.

For SGOCE, my testing was only cursory, and I could not produce a moving robot. Still, the method is retained as an interesting developmental encoding to be explored in the future. The regular cellular encoding is a paradigm similar to the cellular development of our own brains. By embedding the cells in 3D space, the paradigm becomes even more understandable, realistic, and potentially useful.

4.3 NEAT

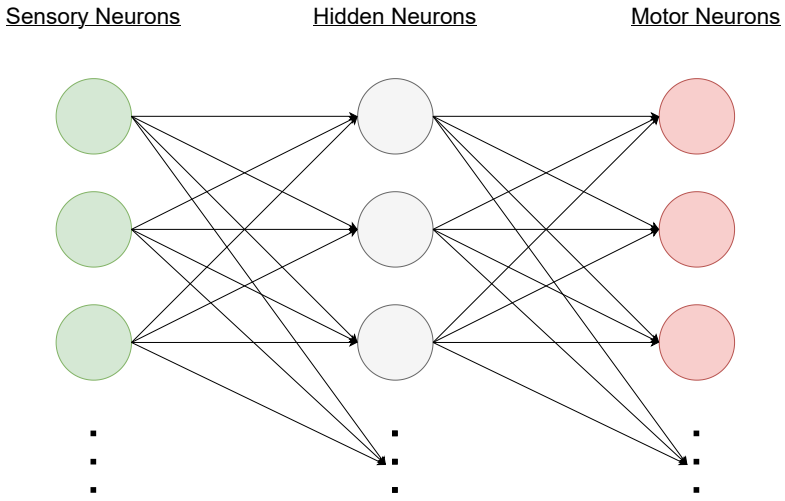

Figure 12: The connectivity of the initial NEAT genome.

After the CE methods, the NEAT algorithm was implemented as described in [Stanley and Miikkulainen, 2002] (though without a speciation mechanism) and tested. Unlike CE, NEAT is a direct encoding, which means it cannot evolve re-usable modules for generating the network structure. Instead, NEAT evolves the brain's structure directly, one neuron and synapse at a time. While direct methods lack in modularity and compactness, one benefit is they allow for fine-tuning the network (at the level of individual neuron/synapse).

Figure 13: The initial NEAT brain.

The setup for the initial NEAT genome was to first add all the sensory neurons to the genome, then to add all the motor neurons to the genome, then to insert some reasonable starting number of hidden neurons (e.g., 128), and finally to connect them as fully connected layers (from sensory layer to hidden layer to motor layer, see *Figure 12,13*).

Mutation operators and rates were experimentally determined. They are as follows:

- **Connections:** Each synapse had 100% chance to be mutated. A synapse is mutated by randomly selecting one of its ABCD Hebbian parameters p_s. There was 97% chance of perturbing p_s by ± 0.05, 0.5% chance of halving p_s, 0.5% chance of doubling p_s, 1% chance of changing the sign of p_s, and 1% chance of disabling the connection (a disabled connection has a 25% chance of being re-enabled during reproduction).
- **Neurons:** Each neuron had 50% chance to be mutated. A neuron is mutated by randomly selecting one of its parameters p_n, either its *bias* or activation function modulator α. There was 99% chance of perturbing p_n by ± 0.05, 0.5% chance of halving p_n, and 0.5% chance of doubling p_n. All neurons used a sigmoid activation function.
- **Other:** No nodes or connections were permitted to be added.

The robots discovered by this algorithm were able to locomote forward rather effectively, though not very "smoothly". Each limb appeared to move independently and erratically, yet the robot was still able to walk forward. This type of behavior is what we would expect to evolve from a direct encoding: since the algorithm is unable to evolve all the legs together, it instead must evolve the subnetworks for each leg separately, and so each leg will move according to its own unique "logic". For a video clip, see *Section 6*.

4.4 HyperNEAT

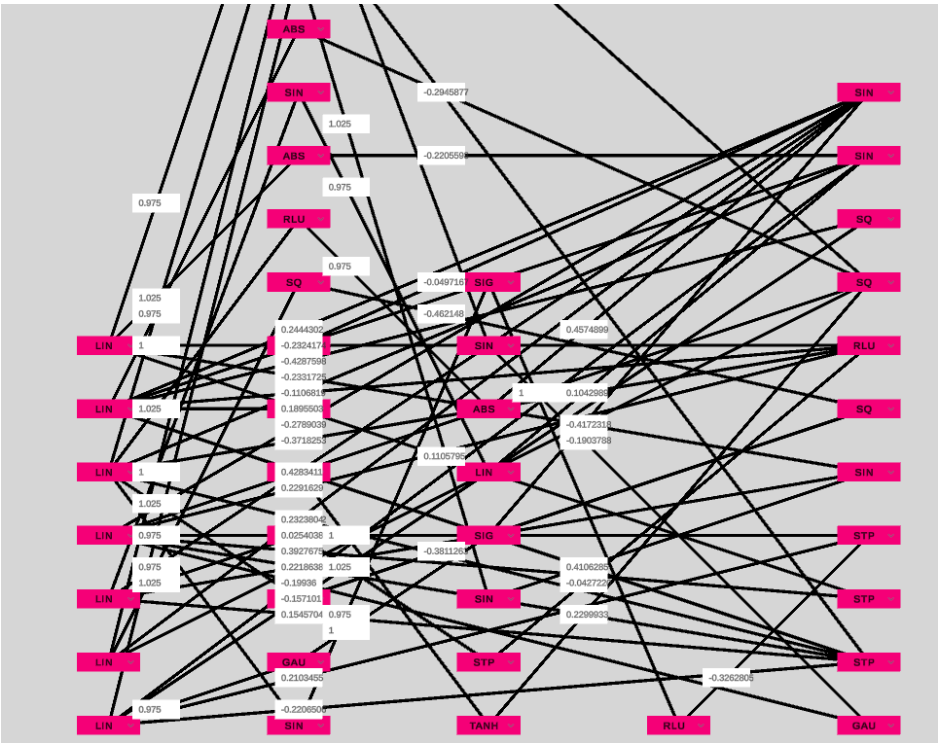

Figure 14: A HyperNEAT genome (i.e., a compositional pattern-producing network) visualized in the *Genome Editor*. The edges point left to right.

The HyperNEAT encoding [Stanley et al., 2009] was implemented in a parallelized ($O(1)$) algorithm. HyperNEAT is an extension of NEAT where, instead of representing the neural network using a NEAT genome, a CPPN is represented using a NEAT genome. The traditional HyperNEAT CPPN for 2D static neural networks has 5 inputs and 1 output.

In HyperNEAT, a neural substrate is first defined by the user, saying how many neurons there are and where they are located in space. It is assumed that the network is fully connected, and the HyperNEAT algorithm determines the connection weights. To generate the neural network weights, the CPPN is queried between all neurons. For example, to determine the weight w on the connection from neuron located at point (x_1, y_1) to neuron at point (x_2, y_2), the 4 coordinates (plus a bias=1) are fed into the CPPN $\langle x_1, y_1, x_2, y_2 \rangle$, and the CPPN's output is interpreted as the value for w. If a connection weight is below a certain threshold, the connection can be disabled (equivalently, setting $w = 0$).

To fit the needs of the *Animat Creator*, the HyperNEAT algorithm had to be modified. Since the animat brains are 3D, an additional z-dimension was added to the HyperNEAT algorithm, by designing the CPPN to take 6 coordinates as inputs rather than 4: $\langle x_1, y_1, z_1, x_2, y_2, z_2 \rangle$. Furthermore, it was extended to evolve connections' Hebbian ABCD parameters in addition to their initial weights, by designing the CPPN to have 5 additional outputs (for the learning rate r, and A, B, C, and D). 4 more outputs were also included for determining α, bias, sign, and neuron activation function. ABCD outputs were scaled by 0.5. The α output was scaled by 0.1. Finally, rather than disabling a connection with a weight threshold, 1 final binary output called the link-expression output (LEO) (introduced by [Verbancsics and Stanley, 2011]) was added, which explicitly determines whether or not to expression the connection between 2 neurons. In total, the CPPN genome for 3D ABCD Hebbian neural networks had 7 inputs and 11 outputs (see *Figure 14*).

The CPPN was initialized fully connected, and stored in a NEAT genome. The nodes in a CPPN were ordered into layers using depth-first search. Then, for a given input, the nodes were evaluated layer-by-layer, starting at the input layer and moving towards the output layer.

Mutation was as follows.

- **Connections:** Each connection in the CPPN had an 85% chance of being mutated. If it was mutated, there was a 97% chance of perturbing the connection weight w by ± 0.025, a 1% chance of halving w, a 1% chance of doubling w, and a 1% chance of changing the sign of w.
- **Other:** There was a 6% chance of changing a random CPPN node's function (to a random function), a 6% of adding a new connection (between 2 random nodes), a 6% chance of adding a new node (at the location of an existing connection), and a 6% chance of disabling a random connection (a disabled connection has a 25% chance of being re-enabled during reproduction).

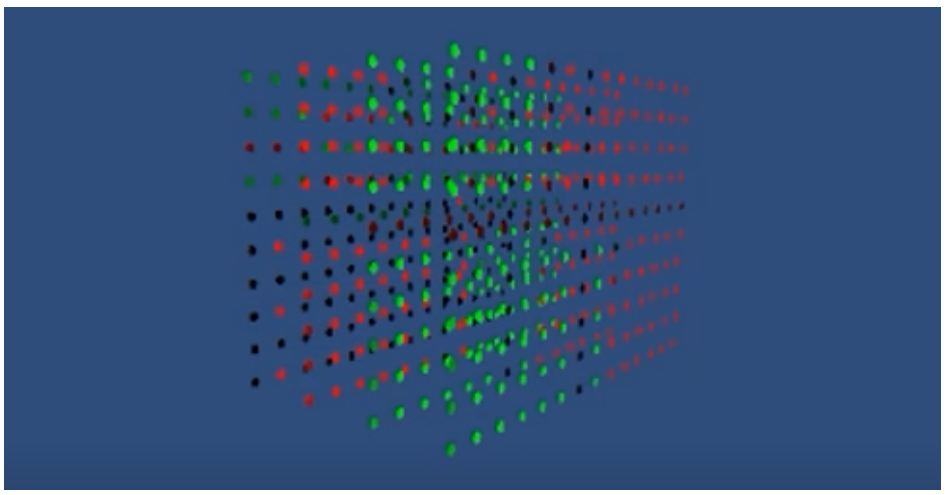

Figure 15: The 4-layer substrate used in the HyperNEAT trials, viewed in the *Brain Viewer*. Notice the geometric patterns of neuron activations.

For the substrate, there were 4 layers: a sensory layer (layer $z = 0$), 2 hidden layers (layers $z = 1$ and $z = 2$), and a motor layer (layer $z = 3$). This was done by instantiating a cuboid substrate of dimensions $14 \times 10 \times 4$ neurons: 14 neurons wide to capture all 14 body segments of a quadruped, 10 neurons tall to capture all 10 sensory neurons in each body segment, and 4 neurons/layers deep.

The sensory-motor layers were laid out in an ordered fashion, though in simple columns, not matching the geometry of the robot's morphology (which would possibly produce better results, though is not strictly necessary). Each leg's sensory-motor neurons were lined up vertically in columns. In a sensory column, the top 4 neurons were the rotation sensors, and the bottom 6 neurons were the touch sensors. In a motor column, motor neurons were equally spaced such that x-drive is at the bottom (on row 0), y-drive is 4 neurons up (on row 4), and the z-drive is an additional 4 neurons up (on row 8). Neurons on the other rows of the motor layer were used as hidden neurons (see *Figure 15*).

The HyperNEAT experiments resulted in some of the most interesting robotic behavior, in my opinion, in the sense of being the closest-looking to animal-like movement. Many of the robots exhibited movements similar to CE, moving rhythmically and somewhat predictably. However, the best HyperNEAT bots moved differently, in a way that looked smooth and rhythmic yet also controlled and deliberate. The best HyperNEAT robot planted its feet on the ground to push its legs and consequently its body up off the ground. Then, while maintaining straight posture, the robot lifted and planted its legs in a coordinated fashion (see *Figure 17*), resulting in what looked like an animal gait or deliberate walk, if rather clumsy. The brain activations as shown in the *Brain Viewer* were also interesting, forming geometric patterns (see *Figure 15*), probably related to the specific CPPN function used to generate the network connectivity. For a video clip, see *Section 6*.

4.5 ES-HyperNEAT

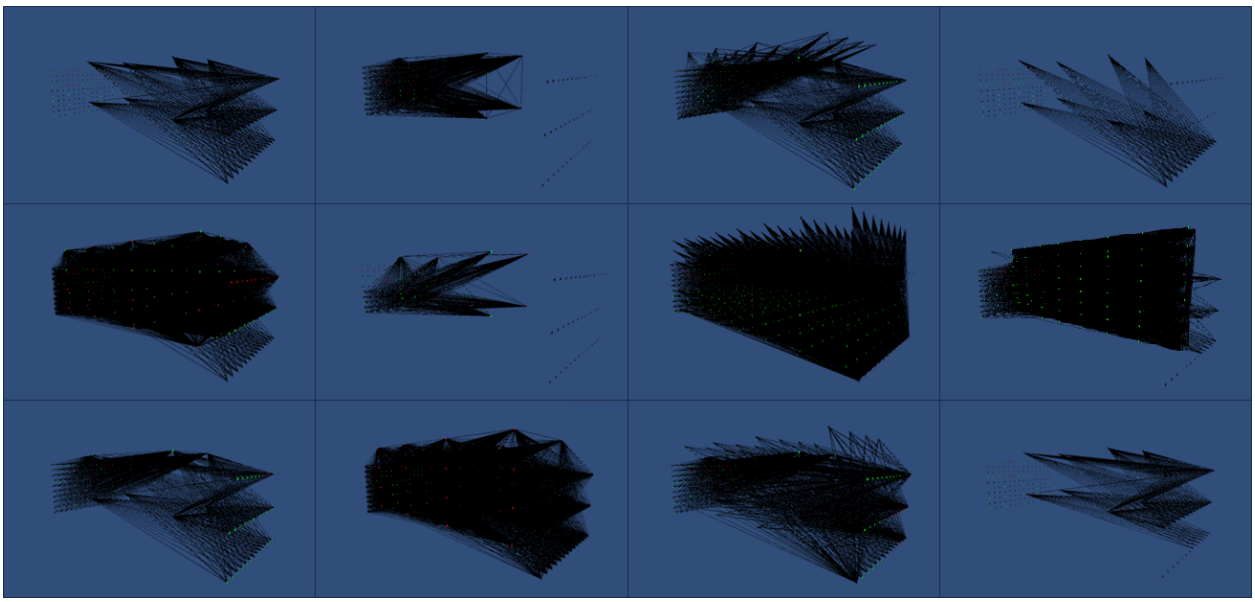

Figure 16: Various brains discovered with ES-HyperNEAT

The evolvable-substrate extension to HyperNEAT (ES-HyperNEAT) was implemented as described in [Risi and Stanley, 2012]. Whereas in the regular version of HyperNEAT, the hidden neurons and their positions are pre-specified by the user, in ES-HyperNEAT the hidden neurons and their locations are evolved, specified by the CPPN pattern. It does not require any additions to the CPPN itself. Instead, a different algorithm iteratively searches for hidden neurons in regions of high variance/information-density compared to other locations.

The algorithm was extended to work in 3D rather than 2D. Thus, it used octrees rather than quadtrees. The variance v for an octree node with value c and mean value of its k children $\bar{c}$ was computed using the

formula from [Risi and Stanley, 2012, p.8], $v = \frac{1}{k}\sum_1^k(\bar{c} - c)^2$. Since there were multiple parameters evolved per connection in addition to the weight w, the value c for a connection was not just $c = w$, but the sum of all continuous-valued connection parameters (i.e., the sum of bias, all 5 ABCD parameters, α, and w).

In the experiments, ES-HyperNEAT discovered an incredible variety different brain structures (see *Figure 16*). The various neural controllers translated to various behaviors, some of which acted similar to robots evolved with cellular encoding, others which acted closer to the robots evolved with HyperNEAT. For those similar to CE, the robot would gallop forward by moving its legs rhythmically forward and backward. For those similar to HyperNEAT, the robot would plant its feet downward and push its body up. For a video clip, see *Section 6*.

5 Conclusion

In conclusion, 5 paradigms of neuroevolution were implemented and tested for articulated robot locomotion. Using the Cellular Encoding, NEAT, HyperNEAT, and ES-HyperNEAT methods, I was able to find neural controllers capable of locomoting articulated segmented hexapod and quadruped robots within very few (only dozens) of generations. Although I was unable to achieve any results for SGOCE due to limited time available for testing, based on the results it seems very likely that, being a more flexible extension of CE, it is also capable of finding successful neural controllers.

The generated robots from all methods showed many similarities to each other, probably due to sharing the same Hebbian brain paradigm, but they also exhibited some unique differences, due to their unique evolutionary history and genetic paradigm. For example, robots generated from the indirect encodings exhibited smoother, more regular movements than robots from the direct encoding.

Though the ability to evolve Hebbian networks for the articulated robot locomotion task is not a new research finding, here the previous findings are confirmed, for highly-segmented robots, and for various neuroevolution paradigms. The new software tools created here are made available for experimenters to quickly plug in and test neuroevolution algorithms on the locomotion task.

5.1 Future Directions

In the future, it would be interesting to try spiking neurons rather than, or in addition to, continuous neurons; as the Hebbian paradigm is based on biological brains, it would make sense to pair it with a model of the natural spiking behavior of biological neurons.

Another factor to consider is the co-evolution of robot body morphology alongside neuroevolution, as was done in Sims' evolving creatures [Sims, 1994]. For example, CPPNs evolved with NEAT have been used to evolve the morphology of voxel robots [Cheney et al., 2014], though without neural control. Naturally, it would be interesting to experiment with brain-body co-evolution using two CPPNs to represent both neural network and body, or perhaps even the same single CPPN/genome for both. This would also facilitate laying out the sensory-motor neurons in a geometric fashion that mimics the body's morphology, unlike I did in the HyperNEAT experiment here, where I simply lined up each body segment's neurons into vertical columns since I did not want to try estimating the neuron locations. Instead, with paradigms combined in the same physical space, the sensory-motor neurons would *automatically* be placed in regions of physical space where the corresponding body segments are actually located. Another option to consider along this line is a body morphology version of SGOCE, where voxels divide in 3D space to generate a voxel robot according to a cellular program. Regardless, the *Animat Creator* should support arbitrary body genetic encodings, and provide an interface able to link them with arbitrary brain encodings, though of course since different genetic encodings will be incompatible with each other, all encodings must at least conform to some minimum requirement or interface to ensure compatibility.

In the future, the *Animat Creator* and its associated tools should be updated to ease testing for the user. For example, a new feature should be added to allow testing genomes in training pods directly from the *Genome Editor*, rather than the current arduous process which requires saving the genomes to disk and then loading them. There should also be a new feature to design body morphology genomes in the *Genome Editor*, to go along with user-designed brain genomes. That way, users can design entire creatures, not only their brains. Then of course, there should be the option to evolve both brain and body genomes together.

In terms of next steps towards the overall research goal set out in [Hahm, 2022] of creating a nature-inspired ecosystem of evolving AI robots, the ability to successfully evolve Hebbian robots which locomote in a certain *absolute* direction is good evidence that these robots can also be evolved which locomote in a certain *relative* direction, that is, *towards* certain objects according to sensory stimuli. In the context of an ecosystem, the minimum required attractive stimuli would be *food* and *mates*. Thus, the next step is to evolve robots which move towards food and mates, then implant the "minimum" creatures into an ecosystem as the starting point for evolution. These behaviors should be sufficient to sustain an active population and evolve interesting diverse species, as evidenced by the populations of Hebbian animats in *PolyWorld* [Yaeger et al., 1994].

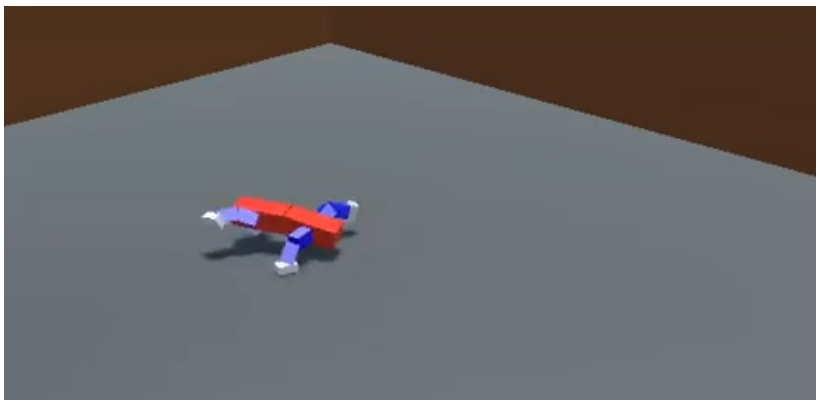

Figure 17: A walking robot.

6 Code and Videos

The *Animat Creator* is open-source and available here: https://github.com/ccrock4t/AnimatCreator
Video recordings of some results are available at these URLs:

- Cellular Encoding: https://www.youtube.com/watch?v=yyDE7w-ajMA
- NEAT: https://www.youtube.com/watch?v=3PsT3HhZ-AA
- HyperNEAT: https://www.youtube.com/watch?v=NbGKKZ56JVA
- ES-HyperNEAT: https://www.youtube.com/watch?v=2ID8hFq6fGA

Acknowledgements

Built with Unity [Juliani et al., 2018]. Thanks to Dr. Kenneth Stanley for answering my questions about CPPNs.

References

[Cheney et al., 2014] Cheney, N., MacCurdy, R., Clune, J., and Lipson, H. (2014). Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. *ACM SIGEVOlution*, 7(1):11–23.

[Clune et al., 2010] Clune, J., Beckmann, B. E., McKinley, P. K., and Ofria, C. (2010). Investigating whether hyperneat produces modular neural networks. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 635–642.

[Clune et al., 2009] Clune, J., Beckmann, B. E., Ofria, C., and Pennock, R. T. (2009). Evolving coordinated quadruped gaits with the hyperneat generative encoding. In *2009 iEEE congress on evolutionary computation*, pages 2764–2771. IEEE.

[Clune et al., 2011] Clune, J., Stanley, K. O., Pennock, R. T., and Ofria, C. (2011). On the performance of indirect encoding across the continuum of regularity. *IEEE Transactions on Evolutionary Computation*, 15(3):346–367.

[Gruau, 1994] Gruau, F. (1994). *Neural network synthesis using cellular encoding and the genetic algorithm*. PhD thesis, l'ECOLE NORMALE SUPERIEURE DE LYON.

[Gruau, 1995] Gruau, F. (1995). Modular genetic neural networks for 6-legged locomotion. In *European Conference on Artificial Evolution*, pages 199–219. Springer.

[Hahm, 2022] Hahm, C. (2022). Designing naturalistic simulations for evolving agi species. In *International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 296–303.

[Hebb, 1949] Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. John Wiley & Sons.

[Hussain, 1997] Hussain, T. (1997). Cellular encoding: review and critique. *Queen's University*.

[Juliani et al., 2018] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

[Kodjabachian and Meyer, 1998] Kodjabachian, J. and Meyer, J.-A. (1998). Evolution and development of modular control architectures for 1d locomotion in six-legged animats. *Connection Science*, 10(3-4):211–237.

[Koza, 1990] Koza, J. R. (1990). *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science Stanford, CA.

[Mill, 2008] Mill, B. (2008). Drawing presentable trees. *Python Magazine*, 2(8):08. https://llimllib.github.io/pymag-trees/.

[Najarro and Risi, 2020] Najarro, E. and Risi, S. (2020). Meta-learning through hebbian plasticity in random networks. *Advances in Neural Information Processing Systems*, 33:20719–20731.

[Risi and Stanley, 2012] Risi, S. and Stanley, K. O. (2012). An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial life*, 18(4):331–363.

[Sims, 1994] Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22.

[Stanley et al., 2009] Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212.

[Stanley and Miikkulainen, 2002] Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

[Verbancsics and Stanley, 2011] Verbancsics, P. and Stanley, K. O. (2011). Constraining connectivity to encourage modularity in hyperneat. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1483–1490.

[Wilson, 1986] Wilson, S. W. (1986). Knowledge growth in an artificial animal. In *Adaptive and Learning Systems*, pages 255–264. Springer.

[Yaeger et al., 1994] Yaeger, L. et al. (1994). Computational genetics, physiology, metabolism, neural systems, learning, vision, and behavior or poly world: Life in a new context. In *SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY-PROCEEDINGS VOLUME-*, volume 17, pages 263–263. Citeseer.