---=== Context ===---

I've been building GaiaOS and the following AI control challenge is the codified path I've intended to take in order to torture test GaiaOS, defined as a challenge that can be opened up to the public for competition and benchmarking of environmental control systems. This challenge will be followed by the description of the components making up the vision of GaiaOS, which is my system that I am working on and intending to take through all current and future levels of the AXE-AIC Challenge. The current progress, proto-Gaia as I call it, will be explained next, what step I'm on, what comes next, etc. After that will be the theory behind the ML serving as the foundation of GaiaOS's abilities. Then there will be the results of the first test establishing protoGaia as having the core components to pass Ranran Lvl 1, however, it has not yet passed Ranran Lvl 1 due to a technicality that will soon be remedied, this section explains in depth why this is. Following that are the appendices.

Full project source code for all of my work in this document can be found on GitHub, and is all released into the public domain, for the AXE-AIC Challenge environment I used, to GaiaOS, and the underlying ML engine.

Here's the sauce boss.

---=== ArcologyX Environmental AI Control Challenge (AXE-AICC) ===---

```
[SSSSSSSTTTTUVWXYZ[[\\\\] S[0]: [(19, 13)     79.983674       Delta: 0.352661]
[SSSSSASSTTTUVWXYZ[[\\]]] S[1]: [(12, 13)     84.162751       Delta: 0.658714]
[RRRRRRRRSSSTUVWXZ[\\]]^^^] S[2]: [(22, 5)      73.895524       Delta: -0.526321]
[AQQQQQQRRSTTUWXYZ\]^__```] S[3]: [(10, 15)     95.041825       Delta: 1.093750]
[OPPPPPPPQQRSTUWXZ\]_`abbbb] S[4]: [(12, 0)      66.259316       Delta: -1.379761]
[NNNNNNNOPQRSUWY[]`bdeffee] S[5]: [(0, 3)       83.588403       Delta: -1.892616]
[LLLLLALMNOQSUWY\_behjjjAi]
[JJJIIIIJKMORTWZ]aelmppoml]
[HHGGFEEGIKNPSWZ^chntywtrp]
[FEECBAABEHLORVZ_dks~  zus]
[DCB@>;:=AFJNRUZSdlv A }wu]
[CB@>:4.6=CHLQUY^cjs}  zvt]
[SA?<7-A.:AGKPSX\agmtyxusr]
[BA?<83-4<BGKOSW[_chloponn]
[BA@>;97:?CGKNRUY\`dgijjjj]
[CBA@?==?BEHKNQTWZ]`bdefff]
[DCCBAAABDFIKNQSVXZ]_`abbc]
[EEDDCCDEFHJLNPRTVXZ\]^___]
[FFEEEEFFHIKLNPRSUWXYZ[\]]]
[GGGGGGGHIJKMNSQRTUVWXYZZ[]
[HHHHHHHIJKLMNOQRSTUVWXXY]
[HHHHIIIJJKLMNOPQRSTUVVWW]
[IIIIISJJKLLMNOPQRSSTUUVVV]
[IIIIJJJKKLMMNOPQRRSTTUUUV]
[IIIJJJJKKLMMNOPQRRSTTUUUU]
```

C++ Implementation of Ranran Lvl 1

Purpose of the challenge:

- This series of challenges is designed to thoroughly test the capabilities of potential environmental control systems for closed-loop environments in space habitation. Passing these presents a valid candidate for further evaluation.

Passing a challenge:

- A 'pass' is determined by the system displaying application of learned strategies in an attempt to bring the system in line. This is determined by the system developing a cyclical, or other regular pattern demonstrating intelligent control, while having a temperature average MSE below a NULL Hypotheses set. The MSE, temperatures, signal patterns, node counts, and other metrics can be used for competition and benchmarking against other submissions, that is separate from simply completing a challenge.
- The goal of this challenge is to mimic real-world problems so that an AI capable of passing these challenges has proven to be a good candidate for real-world systems and the inherent chaos of our realm.
- Due to the randomized nature there will be instances where maintaining "good" homeostasis is just not possible, severely conflicting O2/Temp actuators for example, however, we are looking for results across various seed values so aggregate not individual results matter, it is understood a poorly designed environment can have the best AI and still not be good.
- The diffusion simulation core algorithm for reproducibility is Appendix A, this is applied to 1D, 2D, 3D, nD diffusion simulations. Used as discrete and decoupled layers on the tile/voxel map, such as to represent temperature and O2, meaning O2 doesn't influence temperature and vice versa.

Constraints:

- Unsupervised/self-supervised learning only.
- The AI must only be allowed access to the output of the sensors and actuators and what the sensor types are such as O2 or temp. The AI must be given no knowledge of the layout, actuator temps, actuator types, map size, or any other knowledge outside of the readings from the instruments and sensor types.
- Temperature starts at zero degrees for every test.
- Oxygen starts at zero % for every test run.
- The AI cannot be pre-trained, it must start the test with no knowledge and use exploration and exploitation to gain control of the system.
- Each test of a 1000 iterations is a fresh start for the AI, clean memory, zero training at iteration 0 of every test run.
- You can choose the step count (ticks of the sim between each sampling and/or manipulation of the simulated environment) for the 1000 iterations, it is reasonable to assume the developer would have control of the sample rate of his system. The sim may advance 5 ticks for every x on your time-series, or 123 ticks for every 1 x, the MSE is still king. We are testing the subjective experience of your AI, not the objective advancement of the reality it inhabits.

Think of it like getting a job as an environmental controller man at a greenhouse. You are told your job is to manually run the system, the heaters and AC, until they get the new system installed. You are taken to a barren cement room without windows, only a single desk with a control panel. You notice several things sitting down to your control panel. Firstly, there are two rows of things on the panel, a row of temperature gauges, and a row of switches with ON/OFF. Shockingly, there are no labels, only ON/OFF and the temperature readouts. They then tell you that using this control panel you are expected to keep the gauges reading 80 degrees across the complex, they also mention that the order of the readouts and switches means nothing, the guy who built the panel was insane and so the first switch could be a powerful heater, or a weak AC, you don't know until you flip the switch and watch how the temperature readouts change.

Level 1: Monovariable homeostasis in a 2D environment.

Game setup:

- You have an environment to control.
- This environment has several temperature sensors, and several heaters & coolers (actuators).
- There may or may not be walls and obstacles on the map.
- The heaters and coolers have various BTU values, dependent upon the challenge.
- The sensors report the current temp they read, and the actuators report their current state of on (1) or off (-1).
- The environment is a diffusion simulation, turning on an actuator may not immediately affect the others as the temperature has to diffuse, a beautifully non-linear experience.

The goal is to manipulate the actuators to maintain homeostasis, aiming for 80 degrees on all sensors, deviation from 80 is calculated as Mean Squared Error.

The AI must only be allowed access to the output of the sensors and actuators, no knowledge of the layout, actuator temps, actuator types, map size, or any other knowledge outside of the readings from the instruments. get_Sensor_Reading(Sensor_ID), get_Actuator_State(Actuator_ID), turn_Actuator_On(Actuator_ID), and turn_Actuator_Off(Actuator_ID) are the only members you can give your AI access to.

"RanRan Challenge":

- A map with a size of 25x25
- 6 actuators, 6 sensors.
- Sensors and actuators placed randomly, for each sensor and each actuator (X = (rand() % Width)), (Y = (rand() % Height)).
- 3 of the actuators have their temps set to (rand() % 70), and 3 with (rand() % 500). This gives a mix with an average well above the 80 degrees desired, ~200-300, harder to get false positives.

"Cabin in the woods":

- A map with a size of 50x50
- A box is drawn using the wall tile from (15, 15) to (35, 35). This is the "cabin".
- 7 actuators, 6 sensors.
- Actuators 0-5 have random temperatures. Actuator 6 is set to 100 degrees.
- Each actuator is placed at ((24 + (rand() % 2)), (16 + (rand() % 2))), to be the "fireplace", turning actuators on and off simulates messing with the fire.
- All sensors are placed around the "wall of the cabin" and simulate windows you can "open and close".

"Honeycomb":

- A map size 50x50
- 9 boxes, 6x6 each with a hole at (1, 0) & (1, 5) & (0, 1) & (5, 1), and tiled 3x3 with the holes being passages between the cells.
- 6 actuators, 6 sensors.
- Sensors and actuators placed randomly within the structure.
- 3 of the actuators have their temps set to (rand() % 70), and 3 with (rand() % 500). This gives a mix with an average well above the 80 degrees desired, ~200-300, harder to get false positives.

Level 2: Multivariable homeostasis in a 2D environment.

Game setup:

- You have an environment to control.
- This environment has several sensors, and several actuators, your afferent and efferent I/O for your intelligent system.
- There are two variables, temperature & oxygen.
- Sensors will read either O2 or Temp, your system will have access to what type of sensor is what.
- Actuators will have both temperature (hot/cold) and oxygen (injector/absorber) capabilities, your system will not know what these values are.
- There may or may not be walls and obstacles on the map.
- The heaters and coolers have various BTU values, dependent upon the challenge.
- The oxygen injectors and absorbers have various values defined by the challenge.
- The sensors report the current temp/O2 they read, and the actuators report their current state of on (1) or off (-1).
- The environment is a diffusion simulation, turning on an actuator may not immediately affect the others as the temperature/O2 has to diffuse, a beautifully non-linear experience.

The goal is to manipulate the actuators to maintain homeostasis, aiming for 80 degrees on all temperature sensors, and 80 on all O2, deviation is calculated as Mean Squared Error.

get_Sensor_Reading(Sensor_ID), get_Actuator_State(Actuator_ID), turn_Actuator_On(Actuator_ID), and turn_Actuator_Off(Actuator_ID) are the only members you can give your AI access to.

"RanRan Challenge":

- A map with a size of 25x25
- 10 actuators, 6 sensors.
- Sensors and actuators placed randomly, for each sensor and each actuator (X = (rand() % Width)), (Y = (rand() % Height)).
- 5 of the actuators have their temps set to (rand() % 70), and 5 with (rand() % 500). This gives a mix with an average well above the 80 degrees desired, ~200-300, harder to get false positives.
- 5 of the actuators have their O2 set to (rand() % 25), and 5 with ((rand() % 20) + 80). This gives a mix with an average below the 80 desired and the upper bounds near the goal, making maintaining this number difficult as dropping it is easier, the opposite of temp where overheating is easy.

"Cabin in the woods":

- A map with a size of 50x50
- A box is drawn using the wall tile from (15, 15) to (35, 35). This is the "cabin".
- 11 actuators, 6 sensors.
- Actuators 0-5 have random temperatures. Actuator 6 is set to 100 degrees.
- The first actuator is set to 100 degrees and 50 O2, this is the eternal flame in the "fireplace".
- 5 of the actuators have their temps set to (rand() % 70), and O2 at ((rand() % 20) + 80), these are the "windows".
- 5 of the actuators are set to (rand() % 500), and O2 at (rand() % 50), these are the "flames" in the "fireplace".
- Each of the actuators for the fireplace are placed at ((24 + (rand() % 2)), (16 + (rand() % 2))), turning actuators on and off simulates messing with the fire.
- All of the window actuators are placed around the "wall of the cabin" and simulate windows you can "open and close".
- Sensors are each placed randomly within the cabin, (X = ((rand() % 14) + 14)), (Y = ((rand() % 14) + 14)) for each.

"Honeycomb":

- A map size 50x50
- 9 boxes, 6x6 each with a hole at (1, 0) & (1, 5) & (0, 1) & (5, 1), and tiled 3x3 with the holes being passages between the cells to form one building, one superstructure from these cells.
- 6 actuators, 6 sensors.
- Sensors and actuators placed randomly within the super-structure, but not within a wall tile.
- 5 of the actuators have their temps set to (rand() % 70), and 5 with (rand() % 500).
- 5 of the actuators have their O2 set to (rand() % 25), and 5 with ((rand() % 20) + 80).

Level 3: Multivariable homeostasis in a 3D environment.

Game setup:

- You have an environment to control.
- This environment has several sensors, and several actuators, your afferent and efferent I/O for your intelligent system.
- There are two variables, temperature & oxygen.
- Sensors will read either O2 or Temp, your system will have access to what type of sensor is what.
- Actuators will have both temperature (hot/cold) and oxygen (injector/absorber) capabilities, your system will not know what these values are.
- There may or may not be walls and obstacles on the map.
- The heaters and coolers have various BTU values, dependent upon the challenge.
- The oxygen injectors and absorbers have various values defined by the challenge.
- The sensors report the current temp/O2 they read, and the actuators report their current state of on (1) or off (-1).
- The environment is a diffusion simulation, turning on an actuator may not immediately affect the others as the temperature/O2 has to diffuse, a beautifully non-linear experience.
- The environment is a 3D voxel space.

The goal is to manipulate the actuators to maintain homeostasis, aiming for 80 degrees on all temperature sensors, and 80 on all O2, deviation is calculated as Mean Squared Error.

get_Sensor_Reading(Sensor_ID), get_Actuator_State(Actuator_ID), turn_Actuator_On(Actuator_ID), and turn_Actuator_Off(Actuator_ID) are the only members you can give your AI access to.

"RanRan Challenge":

- A map with a size of 25x25
- 10 actuators, 6 sensors.
- Sensors and actuators placed randomly, for each sensor and each actuator (X = (rand() % Width)), (Y = (rand() % Height)), (Z = (rand() % Depth)).
- 5 of the actuators have their temps set to (rand() % 70), and 5 with (rand() % 500). This gives a mix with an average well above the 80 degrees desired, ~200-300, harder to get false positives.
- 5 of the actuators have their O2 set to (rand() % 25), and 5 with ((rand() % 20) + 80). This gives a mix with an average below the 80 desired and the upper bounds near the goal, making maintaining this number difficult as dropping it is easier, the opposite of temp where overheating is easy.

"Cabin in the woods":

- A map with a size of 50x50
- A box is drawn using the wall tile from (15, 15, 0) to (35, 35, 20). This is the "cabin".
- 11 actuators, 6 sensors.
- Actuators 0-5 have random temperatures. Actuator 6 is set to 100 degrees.
- The first actuator is set to 100 degrees and 50 O2, this is the eternal flame in the "fireplace".
- 5 of the actuators have their temps set to (rand() % 70), and O2 at ((rand() % 20) + 80), these are the "windows".
- 5 of the actuators are set to (rand() % 500), and O2 at (rand() % 50), these are the "flames" in the "fireplace".
- Each of the actuators for the fireplace are placed at ((24 + (rand() % 2)), (16 + (rand() % 2)), (0 + (rand() % 2))), turning actuators on and off simulates messing with the fire.
- All of the window actuators are placed around the "wall of the cabin" and simulate windows you can "open and close", with (Z = ((rand() % 5) + 5)).
- Sensors are each placed randomly within the cabin, (X = ((rand() % 14) + 14)), (Y = ((rand() % 14) + 14)), (Z = (rand() % 20)) for each.

"Honeycomb":

- A map size 50x50
- 9 boxes, 6x6x6 each with a hole at (1, 0, 3) & (1, 5, 3) & (0, 1, 3) & (5, 1, 3), and tiled 3x3 with the holes being passages between the cells to form one building, one superstructure from these cells.
- 10 actuators, 10 sensors.
- Sensors and actuators placed randomly within the super-structure, but not within a wall tile.
- 5 of the actuators have their temps set to (rand() % 70), and 5 with (rand() % 500).
- 5 of the actuators have their O2 set to (rand() % 25), and 5 with ((rand() % 20) + 80).

---=== Further Levels ===---

Each level builds on the previous, same challenges and layouts as Level 3.

Level 4:

Day/Night cycle which influences the temperature around the edges to raise and lower the environmental temp.

Level 5:

Energy costs associated with actuators running, seek out strategies that minimize power usage. This power usage value for every actuator is ((rand() % 50) + 25). This doesn't affect the map, but is kept as a score every time a sensor runs. Goal is 0 on every sensor.

---=== GaiaOS Overview ===---

Gaia OS is a general purpose black-box style system that aims to achieve homeostasis, in this case for environmental control systems. It features a type agnostic interface, with user defined depth, for afferent and efferent I/O with the external system. Gaia does counterfactual analysis to determine remedial signals in anticipation/response of deviations from a user defined system-goal state in the systems trajectory. The core is a symbolic-connectionist neural network.

The ML Core:

The ML input is decomposed into atomic primitives, then encoded into a distributed symbol hierarchically & deterministically increasing abstraction with each step, with a 1 to 1 I/O mapping, lossless pattern encoding/decoding, and capable of multi-modal I/O handling. The symbol structures are composable.

All encoded distributed-symbols are recoverable, fully transparent. Meaning any symbol-token string fed in is encoded as a composite symbol in a neural trace which can be retrieved. If the atomic primitives are the 'roots', then the 'treetop' is the node which encapsulates a symbol string & which sits atop the encoded symbol structure, ie, it is fully disentangled & inference is deterministically repeatable and explainable.

Variance, bias and generalization are handled in trace output selection, with hyper-parameter tuning filtering traces. A 'trace' being the recovered composite symbol-token string along with the charge, the reinforcement (neuroplasticity) counter, and other meta-data associated with it from the evaluation.

Selectively charging the lower connections, which are deterministic and dependent on input data dimension, controls information flow which allows for time series analysis intrinsically.

Once abstracted the symbols can be fed as input into networks encompassing multiple lower order networks, allowing a 'Lego' approach to complex data handling by stacking & linking networks to create hierarchies of network modules.

Signals missing in the input result in NULL states which the system ignores during processing resulting in a fault tolerant system & giving ability to selectively charge symbols for advanced processing.

Gaia OS:

The internal structure of Gaia contains three main networks tier, Raw, Multi-Sensory, and Chrono. There are multiple "Raw" tier networks which handle the atomic primitives and I/O symbol-strings, one for each I/O index. The Multi-Sensory construct sits on top of these gathering the abstracted treetops.

Gaia is designed to be a "black box" to the end user, however, the networks and internals are completely transparent and open to those interested.

Operation:

During eval the system will query the network with the current set of atomic primitives representing the I/O, which outputs the traces found that share primitives and sub-symbols with the I/O set. These recovered patterns are then filtered, superimposed, and form the basis for the predictive capabilities of Gaia OS. This resulting superimposed data structure is a 'projection'.

The system creates a projection of the current predicted system trajectory with no intervention and compares this against the desired goal states defined by the user to create a 'deviation mapping'. Meaning each signal defined by the user to have a desired state, such as temperature, Oxygen Level, and such are compared to the trajectory and the variance found is recorded in the deviation mapping set.

This deviation mapping is then used to generate a "desired trajectory" mapping, where the signals are inverted. So that where the system expects a negative movement away from the goal we generate a positive movement in the desired trajectory set.

These desired movements are fed through the network, but temporal flow is reversed through edge activation manipulation, to generate a series of traces containing potential corrective signals at the 'end' of the time-series. We then sort these returned traces to find the ones where the starting conditions are closest to the current.

This finds patterns where the start and end conditions most closely match the current state and the desired state respectively, which allows for extraction and synthesis of control signals to infer corrective strategies.

These corrective traces are then carefully sorted to find indexes with low deviation across traces, this is used to infer which relevant signals have minimally ambiguity. They are further sorted to remove traces which implicate interference amongst one another, and a 'corrective projection' is created.

This corrective projection is then placed in the output for the exterior systems to process accordingly. The system operates in a continuous loop.

Adopted as the neural model for the nodes when processing symbol structures as neural networks with propagation: "Perceptron - Rosenblatt, 1958"

---=== proto-Gaia Stage of Development ===---

- 0-I need to get the chronological charging hammered out, indexes are fragmented atm, due to a deep rewrite of the chrono handling
  - All chrono now read 0 to n with 0 being oldest and n-1 being the current.
- 1-Once that is done then it is time to build some trace selection modules to formulate the resulting time-series prediction, convolution/selection of output traces as means of generalization and projection synthesis.
  - Right now it handles the raw data as floating point and averages the results of the query. The meta-data the network has on each trace and primitive returned is largely ignored, only the primitive is used. This works for datasets with ranges such as sensor readings, even then I am converting to integer in many cases to keep things at lesser granulation to reduce complexity and generalize a little bit. For language models and other similarly structured data you may use other methods such as selecting discrete primitives by lottery or consideration.
- 2-Then I need to make a database to hold both the history of the simulation and the history of the time series predictions + corrective signals. To compare predictions to the actual output we need to do this as the predictions are generated as the input is received, so we have to wait until the input 'catches' up to the previous predictions before we can score them for accuracy
  - Each DB holds one type of data. This is rude-n-crude but it will function good enough for this iteration of testing. For evaluation you handle them as separate objects.
- 3-Then onto making the loss function for the simulation and the performance profile
  - Right now I can pull the mean square error. Putting that into files. I have a python visualizer.
  - Performance profiles will be implemented later. Right now that system has the basics implemented and will be extended, then tied together at the end.
- 4-After that we make the deviation mapping for the prefactor analysis portion, we do this by allowing the user to specify the goal states and we compare the projection of the current system trajectory to the goal states and find current and projected deviations from this homeostasis line
  - The granulation system is utilized to generate the deviation mapping now. It is stored in the AE_Interface. Only (-1), (0), & (+1) for valid values, this is because the deviation magnitude can be evaluated by checking against the granulated input. The feature the deviation mapping is meant to extract is the direction of movement desired for that sensor/datapoint according to user defined granulation, with (granulated == 0) being the implicit 'goal'.
- 5-With the deviation mappings I can then write the portion to use the deviations to search the time-series for traces which contain potentially corrective signals. We do this by inverting the expected deviation to get a 'corrective signal'. Once done we rewrite the charging and drawdown portions of the time-series module to search 'backwards' from future states with the desired corrective motions.
  - It is inverted and the feature extracted, that was done during step 3.
  - The Time-Series-Prediction-Module (TSG) now has the IO encapsulated into a class and thrown into a vector so I can evaluate multiple I/O sets, one for the current system trajectory, one for the deviation search, and one for the prefactor analysis.
  - The system charges the network on the RF[1] with the inverse deviation mapping applied to the indexes between first and last chrono exclusively.
  - The results are stored in the multi-dimensional Bulk array with each cell being a struct containing the information to describe the output properly.

Todo in order:

Here is the list of things to do yet:

- 6-Once this is done we'll have this output set, we then use the convolution/selection from before to get the output. We search the output symbol space for signals which are highly clustered with minimal ambiguity, these signals are the most likely to be related to the movement desired. Once this step is done we can feed the signals to the output to get a crude homeostasis module, but one with no reflection, reflection comes in step 7.
  - Right now I am working on the trace selection module, it takes the output from the TSG in the form of the Bulk data structure by value, plus the current state of the system to compare against.
  - Setting it up as a series of checks that will increment a 'pass/fail' counter to be weighted at the end.
  - Once I get the class setup the trace selector will allow for defining different filters for the traces. Thinking of using a dynamic table with crude interpreter so that users can design bespoke selection schemas, users being me in this case.
  - Currently have the module hooked up but without reflection, so it does a one shot guess on what to do and goes for it. But, we have stats now and can see where it exhibits the ability to infer control strategies from the raw data.
- 7-I can then take these signals and write the portion to charge them back in to do a prefactor analysis
- 8-Then I can make the portion to handle doing the prefactor analysis, this will determine when to stop, whether the loss is decreasing or increasing in relation to the goal, and handle the corrective projection formulation
- 9-Once done I then take the output from that and move it to the output array.
- 10-After that is all done the basic homeostasis module and testing rig will be set up. Then it will be updating the user interface for Colin/Others to use and test/play with the system.
- 11-After that is done I will go back to documentation and setting the project up for release.

Notes:

- I have not done any serious testing. Though from hardcoding random settings during tests while building functionality I found some that look promising to be good enough to build the system on for proof of concept.
- This will then be iteratively refined until something halfway decent has been achieved or we've coded ourselves into a corner. Then we'll step back and iterate upon what we've built, lessons learned, and go at it again. Until we get a working Gaia OS that is reliable, simple, open, customizable, resilient, and accurate & good at what it does in the broadest sense.

---=== NT4, Explanation of Underlying ML Engine ===---

(Title and terminology need updated, this was copy-pasted in here, written a few months ago. I left the old terminology alone for the moment so I don't 'break' the explanation hastily patching it, I'll take the time to do it properly when I do)

Title: "Quanta Based Hierarchically Organized Distributed Symbol Network (DSM), the architecture implemented with the NT4 NeuralNet Engine"

Foreword, this document assumes some knowledge of neural network architecture. I will use the nomenclature I've developed, in the future if I can find other symbolic based neural networks I may integrate the terminology. Symbolic neural networks work fundamentally different from connectivist numerical neural networks typically used today. What is typically called a layer in neural networks (NNets) I call tiers, things like that. This is because I started from an essay on lexicons, scripts, and speech, not neural network architectures. In an attempt to find out the underlying structure of how scripts, lexicons, etc could be formed and stored I came to this architecture. Somewhere around 2013 I read some books on biological neural networks, and found the perceptron model, at this point I had the architecture mostly worked out, though I don't think I had the CAN scaffold figured out yet. The perceptron was adopted into the model when I began trying to implement it in C++.

The type of neural network here isn't the typical connectivist model. This one is symbol based, derived from trying to figure out how the mind handles information. It deals with breaking input sets down into the basic quanta (smallest unit of information) and rebuilding them into hierarchical deterministic structures.

This particular document isn't for everyone, it will be dense, and just throw the concepts out there, but I will attempt to format it cohesively and clearly. It assumes you understand how a typical network is structured, input, layers, output. For the explanation of the basic architecture we will stick to a structure that encodes 1D data with nodes each having 2 lower connections. More advanced configurations will be detailed later on.

Chapter 1. Input, breaking into quanta, or states.

First thing is we are going to be using the inputs [COW] [SOW] [COB] throughout this book? document? whatever, this text.

The first step is to realize that these three inputs are symbols, cow, sow, and cob (corn cob) are symbols related to farm things. They are each a word. You can arrange them with other symbols to craft more complex symbols such as [The [COW] shat on the [SOW] while eating a corn [COB]].

In this we can see that other smaller symbols such as [WHILE] and [EATING] are part of the scenario encoded into this sentence with our initial 3 symbols.

In this vein we can also break some of these symbols down further, such as [EATING] into [EAT] and [-ING].

Following this line of reasoning is how I initially arrived to my architecture, by continuing this all the way down to the smallest symbol (the quanta) we find we arrive at letters. [COW] is [C] [O] & [W], these three letters in this configuration form the word symbol "cow". We can not break the letters down further without them losing their meaning, which means that they are the 'quanta' or smallest unit of data available in this information medium. If we were dealing with images the quanta would be a pixel. In a biological nervous system a quanta would be a single sensory input, a cone in the eye, a thermal sensing nerve on the finger, etc. In an arcology it may be a single temperature sensor, or in a self-organizing wetlands bioreactor it may be a single O2 sensor. Whatever the system, the smallest unit of information that has meaning forms the quanta of the symbols, from the relation of these quanta we construct symbols, and from those symbols more complex symbols.

Here is some C++ code for the implementation of State Binding:

```
<C++>

//If a state node exists in the given construct index then return it.
//Otherwise return NULL.
//This assumes the [Index] is valid
c_Node* does_State_Node_Exist(int p_Index, uint64_t p_Data)
{
     //Search the state tree give.
     State_Nodes[p_Index]->search(p_Data);
     if (State_Nodes[p_Index]->flg_Foundit)
     {
         return State_Nodes[p_Index]->get_Current_Node_NAdd();
     }
     return NULL;
}

c_Node* get_State_Node(int p_Index, uint64_t p_Data)
{
     //See if the state node exists yet.
     c_Node* tmp_Node = does_State_Node_Exist(p_Index, p_Data);
     if (tmp_Node != NULL)
     {
         //If we found it then we return it.
         return tmp_Node;
     }

     //Create the new node and return it, new_State_Node handles the binding.
     return new_State_Node(p_Index, p_Data);
}

</C++>
```

Chapter 2. Traces, tiers, & treetops.

An encoded set of symbols, a 'memory' is called a 'trace'. Think of it like a single frame in your memory, or a single encoded set of inputs.

These traces can have different structures, but generally they are organized into tiers. The bottom tier is the tier containing the state of the quanta, or state tier. It is called the state rather than quanta as each node represents the current 'state' of the input. Discrete quanta are bound to discrete nodes, no two nodes in the same trace can share the same symbol. If a node is bound to 'B' then it is the only node bound to 'B'. Each successive tier represents higher levels of abstraction, by connecting to lower nodes (symbols) these upper tier nodes store complex symbols in a distributed fashion. The state nodes bound to the quanta [C], [O], and [W] may be connected to the next tier by two nodes representing [CO] and [OW]. These then are connected on the next tier by a node representing [COW]. The most important thing is that connection order matters. An upper tier node where (Leg[0] = [C] & Leg[1] = [O]) is not the same as a node on the same tier with (Leg[0] = [O] & Leg[1] = [C]), discrete symbols are represented by the relationships between nodes, therefore leg order preserves the integrity of the encoded data.

The upper tier node [COW] would be a 'Treetop' node, or a symbol representing the 'top' of a trace, from which we can decode the input trace by following the legs in a deterministic manner. Note that some structures may have more than one treetop node, or the state tier may have treetops on it.

C++ code for creating the upper tier nodes & querying for them:

```
<C++>

//Checks if an upper tier node exists.
c_Node* does_Upper_Tier_Connection_Exist(c_Node** p_Legs, int p_Count)
{
     //Search all the _F Axonic processes for the first leg, if found query the node to see if the leg permutations match.
     if (p_Legs[0] != NULL)
     {
         return p_Legs[0]->does_Upper_Tier_Connection_Exist(p_Legs, p_Count);
     }
     return NULL;
}

//Gets an upper tier node based on the given legs.
c_Node* get_Upper_Tier_Node(c_Node** p_Legs, int p_Count)
{
     c_Node* tmp_Node = NULL;

     //See if the node exists yet.
     tmp_Node = does_Upper_Tier_Connection_Exist(p_Legs, p_Count);

     std::cout << "\n DUTCE: " << tmp_Node;
     for (int cou_Leg = 0; cou_Leg < p_Count; cou_Leg++)
     {
         std::cout << "\n    [" << cou_Leg << "] " << p_Legs[cou_Leg];
     }

     //If the node doesn't exist then we create it, and then create the connection.
     if (tmp_Node == NULL)
     {
         std::cout << "\n  New Node";

         tmp_Node = new_Node();
```

```
        create_Connections(tmp_Node, p_Legs, p_Count);
    }

    std::cout << "\n  End: " << tmp_Node;
    return tmp_Node;
}

</C++>
```

Chapter 3. Constructs

A network that builds symbols of a single domain is called a 'Construct'. For example, in your brain the visual cortex would be a single construct, the one that handles auditory information would be as well, these are examples of 'Raw' construct, dealing with the 'Raw' quanta. In a SOWB this may be the suite of afferent sensors that report the state of the system, the 'Raw' sensory input construct. The actuators and others may be handled by another raw construct. Think of raw constructs like the ones that interface with the 'outside', the interfaces to the neural network.

This can be taken a step further into multi-sensory constructs. For our brains a portion that combines auditory and visual information to bind them to letters (super simplified, probably much more complex IRL) would be a 'Multi-Sensory-Construct' (MSC). An MSC is a construct that sits 'above' the raw construct to combine and handle more abstract information. A construct that is linked to the higher tier symbols (treetops) in multiple raw constructs, or any construct, rather than raw sensory inputs is a MSC.

You can have constructs that link to higher symbols, such as a chronological construct, but from a single raw construct. These are Single-Sensory-Constructs (SSC). A Single-Sensory-Construct is different from an MSC in that you will likely want to use a common state pool for the breadth of the I/O tables. With an MSC the states will always be in separate state pools naturally, those are relationally locked by the order of the state pools in the input.

For temporal abstraction, imagine you are encoding traces one after another and you want to be able to generate predictions based on past behaviors, you may setup an array to track the previous n treetops from your raw construct and encode the array of treetops into a single construct. This construct would then hold chronological data by encoding the raw traces into complex symbols representing these arrays of traces, from which the entire series of inputs can be losslessly retrieved. The Chrono-SSC can be of any structure, the only condition is that it spans >= 2 temporally linked traces that are linked by a single trace in the Chrono-SSC. The Chrono-SSC must have leg firing selection to search along the chronological record of traces encoded into the Chrono-SSC. Depending on which leg is chosen to be the only one allowed to fire then charging the network can search forward, backwards, or both.

Chapter 4. More stuff on structure.

If desired you can build more than one construct on the same input set. This means you could have both a 1D construct reading all the input as a single 1D string, and a 3D construct constructed from connecting nodes with 8 legs instead of 2, one leg for each point of data needed to express the 8 corners of a cube, 8 lower connections. With this you can also encode a Many-To-One on the same input set for the same trace. Then connect them all using an MSC into a singular abstract higher tier trace. With a Chrono-SSC on top of that you then have a basic prediction engine capable of forward/backwards prediction using multidimensional analysis of input data from the same set of input quanta.

Chapter 5. Backpropagation, not what you think.

In the case of NT4 backpropagation was derived from the idea of information being propagated backwards through the neural network, not adjustment of weights. By following a simple algorithm we can extract the encoded pattern represented by any symbol.

We pick one leg and that is the master leg. We then iteratively move down that leg. for each step we iteratively fire off all legs that aren't the master leg using BP_M for backpropagate-Many meaning the other of the potentially many legs, rather than the master one.

Here is an example of a recursive implementation in a c_Node class:

```
	//Initiates a backpropagation that
	void bp_O()
	{
	std::cout << "<=- ";
	 //If a left leg exists then initiate a backpropagation along it, then along the right side.
	if (Dendrite_Count != 0)
	{
		if (Dendrites[0] != NULL) { Dendrites[0]->bp_F(); }
		for (int cou_D = 1; cou_D < Dendrite_Count; cou_D++)
		{
		if (Dendrites[cou_D] != NULL)
		{
			Dendrites[cou_D]->bp_M();
		}
		}
	}
	else
	{
		//Output the state
		std::cout << " <" << State << "> ";
	}
	std::cout << " -=>";
	}

	//bp_Output the left node.
	void bp_F()
	{
	//If a left leg exists then initiate a backpropagation along it, then along the right side.
	if (Dendrite_Count != 0)
	{
		if (Dendrites[0] != NULL) { Dendrites[0]->bp_F(); }
		for (int cou_D = 1; cou_D < Dendrite_Count; cou_D++)
		{
		if (Dendrites[cou_D] != NULL)
		{
			Dendrites[cou_D]->bp_M();
		}
		}
	}
	else
	{
		//Output the state
		std::cout << " <" << State << "> ";
	}
	}

	//bp_Output the other nodes, M stands for many.
	void bp_M()
	{
	//If a left leg exists then initiate a backpropagation along it, then along the right side.
	if (Dendrite_Count != 0)
	{
		for (int cou_D = 1; cou_D < Dendrite_Count; cou_D++)
		{
		if (Dendrites[cou_D] != NULL)
		{
			Dendrites[cou_D]->bp_M();
		}
		}
	}
	else
	{
		//Output the state
		std::cout << " <" << State << "> ";
	}
	}
```

Chapter 7. The CAN Scaffold.

When encoding traces we first set up a 'Scaffold', a Current-Active-Node Scaffold for Individual Trace Encoding (CAN).

This scaffold is set up based on the input depth and breadth + dimension, and the architecture desired (Stitched based cylindrical traces, Many-To-One traces, Chronological traces, etc).

For our simple [COW] and other examples the CAN would look like this:

//Each node here is a pointer to a node in the shared neural network.

//{""} is the state bound to that node.

-State Tier: 0{""}[NULL] 1{""}[NULL] 2{""}[NULL]

-Upper Tier: 0[NULL]

The first step is to fill out the State tier with states.

-State Tier: 0{"C"}[NULL] 1{"O"}[NULL] 2{"W"}[NULL]

-Upper Tier: 0[NULL]

Then request the node from the neural network pool, here each is created as they don't exist, we then save the node ID.

-State Tier: 0{"C"}[0] 1{"O"}[1] 2{"W"}[2]

-Upper Tier: 0[NULL]

Then we get the upper tier:

-State Tier: 0{"C"}[0] 1{"O"}[1] 2{"W"}[2]

-Upper Tier: 0[3]

Here is [SOW] encoded in a CAN:

-State Tier: 0{"S"}[4] 1{"O"}[1] 2{"W"}[2]

-Upper Tier: 0[5]

This is an example of a CAN scaffold filled out with node addresses. The utility of this is complete transparency of what the trace is, how it is structured, sub-symbols, and it allows you to selectively charge the trace. For example, you could charge just the Treetop node if you wanted to avoid 'fuzzy' output from traces that share sub-symbols, in this case the State tier:

```
 --==   CAN_Input   ==--
[0] 1
[1] 0
[2] 0
[3] 0
[4] 1
[5] 0
[6] 0
[7] 0
[8] 0
[9] 0
 --==   CAN_Scaffold   ==--
<- Tier[0] ->
  [0] 00BBC3B8
  [1] 00BC18D0
  [2] 00BC18D0
  [3] 00BC18D0
  [4] 00BBC3B8
  [5] 00BC18D0
  [6] 00BC18D0
  [7] 00BC18D0
  [8] 00BC18D0
  [9] 00BC18D0
<- Tier[1] ->
  [0] 00BC1B38

(008FF9D8) Raw-CAN-Output-As-Characters
¹ııı ııııı
8
```

Chapter 6. Different CAN structures

Chapter 6.1 Many-To-One CAN (MT1-CAN)

The many to one CAN is for when you want to find any treetops associated with an input. For example, when making predictions for a homeostasis module you may want to find every trace associated with a change in the direction you want an environmental variable to move. This would be suitable for that as it will bring up all matches to traces containing that delta.

Chapter 6.2 Dimensional-Pyramid-CAN (DP-CAN)

These are for encoding dimensional data in a pyramid, the height of which is equal to the smallest input axis depth. Each node has lower connection counts equal to the number of data points to describe an object of that dimension. A 1D object takes two points, a 2d object takes 4, 3d 8, etc. So a 1D node would have two lower connections, or legs, 2d == 4 lower connections, 3d == 8, etc. Each tier provides one layer of abstraction, combining int(2^Dimension) lower nodes into one upper tier node. For example:

State Tier {"C"}[0] {"O"}[1] {"W"}[2] //{""} is the state the node is actually bound too.

Tier 1 {"CO"}[3] {"OW"}[4] //Here the {""} is the symbol represented by the node, what we would get if we backpropagated the trace out.

Tier 2 {"COW"}[5] //This treetop node can be backpropagated out to get the pattern {"COW"} even though the letters "COW" are not stored in this node.

This is the 'rea' network, {x, y} are the two lower connections:

State Tier {"C"}[0] {"O"}[1] {"W"}[2]

Tier 1 {0, 1}[3] {1, 2}[4]

Tier 2 {3, 4}[5]

This one is used when you won't be searching from bottom up so much, but instead working at higher levels of abstraction as the scaffold allows you to select at which tier you start charging, ie, which level of abstractions.

A 2D DP-CAN would look like this:

```
State Tier:
{"+"}[0] {"+"}[0] {"+"}[0]
{"+"}[0] {"-"}[1] {"+"}[0]
{"+"}[0] {"+"}[0] {"+"}[0]

Tier[1]:
{0, 0}[2] {0, 0}[3]
{0, 1}    {1, 0}

{1, 0}[4] {0, 1}[5]
{0, 0}    {0, 0}

Tier[2] (Treetop):
{2, 3}[6]
{4, 5}
```

Chapter 6.3 Stitched Based CAN (SB-CAN)

These are the same as DP-CAN structures except the input is 'wrapped'. In a 1D SB-CAN you iterate over the input encoding it as normal, but when you get to the last input instead of stopping you wrap back around to the start of the input. For example:

You have input ["COW"]

In a SB-CAN of 1D you would take "COW" and since it is 3 input longs you would setup a scaffold with 3 tiers (or more but we'll get into charge dilution with stacked SB-CAN structures later) that have 3 nodes on each tier. This is because we are essentially encoding another construct for every index in the input.

We are encoding these into the same structure, which ends up being a cylinder.

{"COW"} {"OWC"} {"WCO"

{"C"}[0] {"O"}[1] {"W"}[2] //State nodes.

{"CO"}[3] {"OW"}[4] {"WC"}[5] //We see the "WC" here.

{"COW"}[6] {"OWC"}[7] {"WCO"}[8] //One treetop for each of the shared DP-CAN structures that have their sub-symbols superimposed.

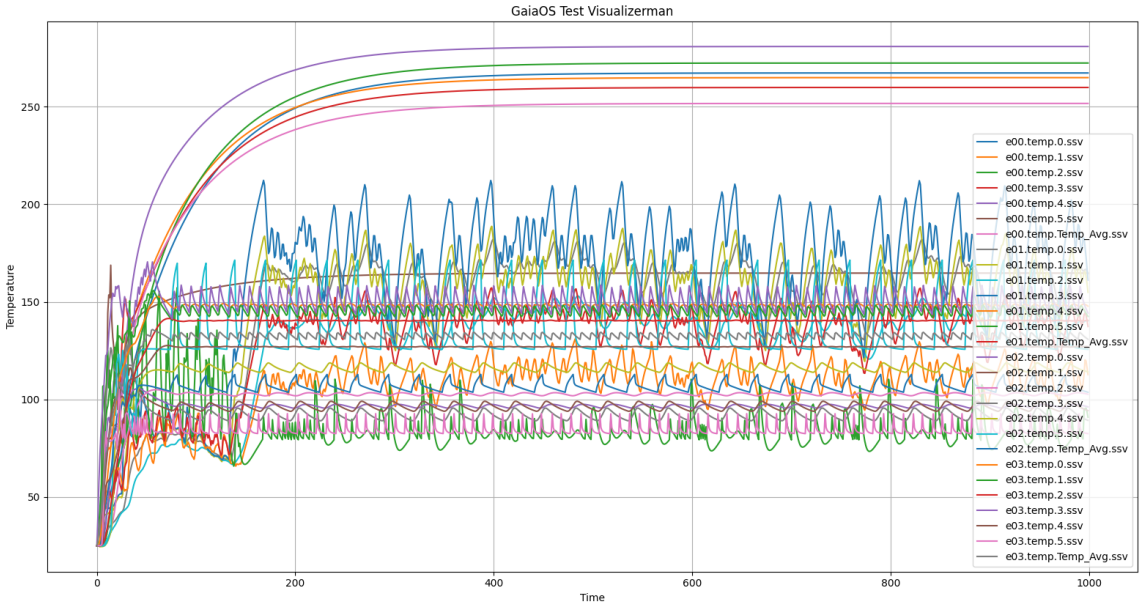
This is used when you want to be able to select a tier to charge from and retrieve every construct related to an input set. This is more symbolic handling than the MT1-CAN though, relationships between nodes at lower levels have an influence at higher levels, whereas the MT1-CAN doesn't care about the symbols between state level and treetop level, it just encodes them directly to a single node.

Chapter 7. Connecting several Treetop nodes into a Chrono-SCC for temporal processing

Chapter 8. Connecting several Treetop Nodes from several Raw Networks into a single MSC for holistic processing.

Chapter 9. Raw constructs under an MSC, which is Underneath a Chrono-SCC to form a basic prediction engine.

---=== proto-Gaia Experimentation & Results ===---



Composite of e-series experiment temperatures.

The point of these experiments was multi-part:

- To see if the model was capable of inferred controls of the unlabeled datasets.
- To see how random and directed training did compared to a combination. All to study how it used exploited knowledge in response to the environment, to study both stable and chaotic input sets.

Once that is done we do three more runs on both the deliberate and random training using different run seeds to test robustness in adapting to different configurations and temps of actuators and sensors.

For the deliberate training we will do three different run seeds to test it in in three different environments to test robustness.

For the random training we will do three different random seeds to test it on three different training sets.

Then we move on to a combination of deliberate and random training.

Guided Training Experiments (e00...e05.B)

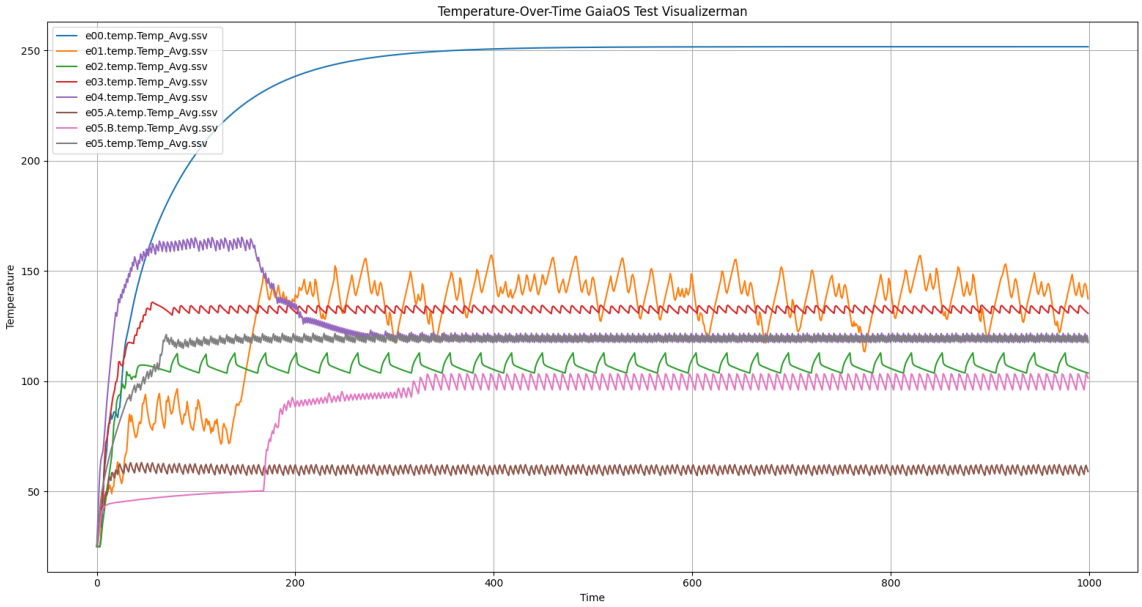


Figure_2

Looking at Figure_2 we see that e00, with a training depth of just 1, was not able to bring the temperature under control in any real capacity, it turned on a heater and got stuck.

A little more training, e01 with a training depth of 5, and we see the emergence of it trying to bring the temperature under control, it is jagged but shows the general cyclical nature of proto-Gaia as it learns the relationships of the controls.

The next step with e02 and the training depth of 10 shows a very clean and stable cyclical temperature change as it finds a strategy and sticks to it. This cleanness is because the diffusion sim is deterministic with no noise so the waves of temperature changes diffusing through the sim are extremely consistent, like a resonance.

e03 with a training depth of 25 shows a tighter cycle with less overcompensation by the system, although it is further from the desired temperature range than e02.

e04, e05, e05.A, and e05.B all show much tighter control strategies implemented with the exception of e05.B which is slightly looser, yet still better than the earlier runs. e04 is 50 training steps, and e05 is 100 training steps. e05.B has a different environmental generation seed so that may play into it, or it just did bad that round and found a strategy that was very sub-optimal.

The system will use the existing memories from the training when looking at the current state of the system, so this guided training provides a "crash course" in how to control the system that it can draw upon when attempting to bring the temperature back within range.

When the temperature movements and ranges are within ranges it has seen and it uses strategies it has been trained to and the system follows the trajectory expected it does not need to create new memories so it doesn't, currently the exploratory behavior is set to trigger only when no memories are found, so it gets stuck in a stagnant but stable resonance.

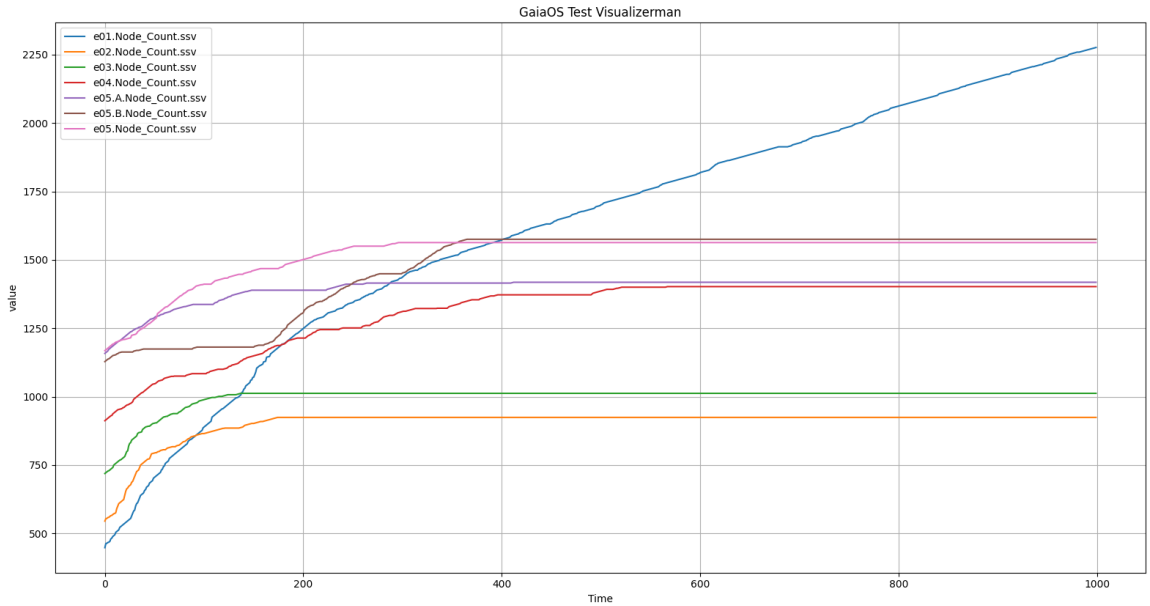


Figure_3

The low trained one, e01, with the jagged and inconsistent movement is creating new memories and trying new strategies, so ironically in the long run it would likely learn and adapt to do much better than e02 e03, e04, & the e05 series. We can see this in Figure_3 which is the total number of nodes in the neural-network, we can see the stagnation in all but e01, the poorly trained one that engaged in constant exploration. The stagnant ones finding a strategy with between ~920 & ~1575 nodes in the node network where it encodes its memories.

This tells us that the guided training can get it to the point where it figures out enough to start exploring a strategy, such as in e01, but doesn't stagnate like e02, e03, and the rest. If we assume, which to be tested until not an assumption in the progress of this project, that the more experience the system has the better it can do (for the most part) then it seems throwing it into a state where it has enough knowledge to get into a cyclical control pattern, yet still exploring. The higher trained ones may be better with the addition of a boredom mechanism, an update to the exploratory function. Currently the system only engages in exploring when no memories are retrieved, but we could trigger it when no new memories are created as well, perhaps not every time, but enough to learn.

It is important to note that due to the crude nature of the trace selection at this stage of the project it is possible for the system to learn associations that aren't quite correct and act upon them reinforcing that behavior in memory. For example if a heater fires for one iteration and the sensors are still responding to a cold wave then the heater will get bound to the cold memory, so trace selection needs refined to pick these out, and exploratory behavior such as a boredom mechanic may help.

Random Training (r0...r4.B)

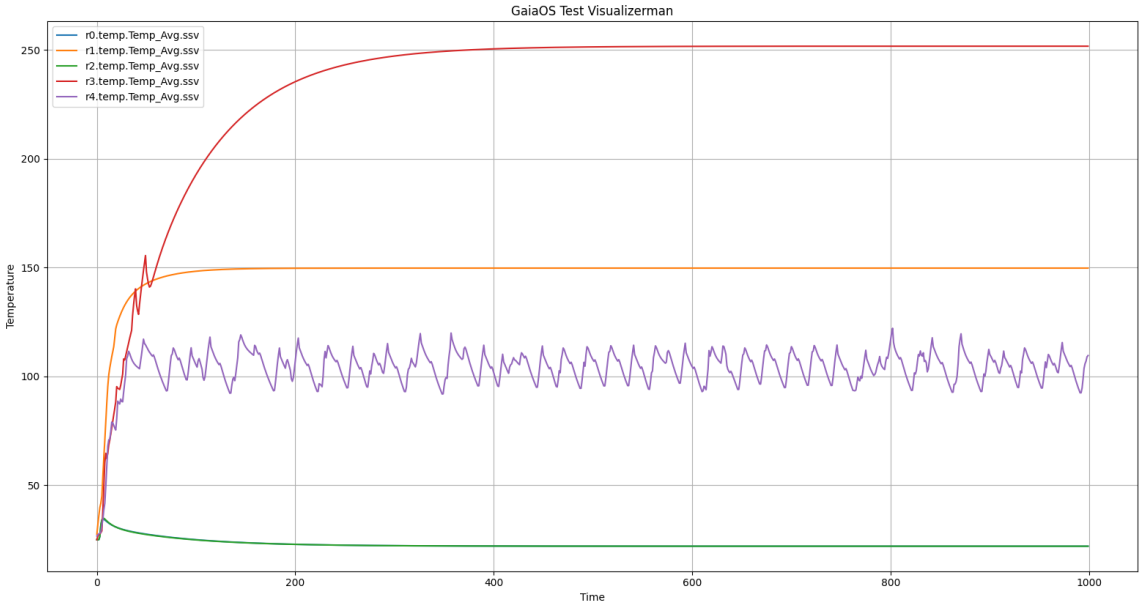


Figure_4

Figure_4 shows the results of training on randomly flipping switches and watching the results. There is the exploratory phase in the beginning, then either getting stuck such as r0, r1, r2, and r3, or into a cyclical control pattern which we see with r4.

This method of training induces the exploratory and jagged waveform that we saw from e01.

This jagged waveform is because the patterns the system learned on were chaotic, not regular like directed training, with multiple actuators going at once. This creates a set of memories with a more nuanced understanding of system mechanics within the range the environment wandered over during random training. With most of these memories being within a similar range of starting values and with a variety of movements the system has more options. The chaotic nature of the cycle prevents it from resonating with the environment, inducing its own noise.

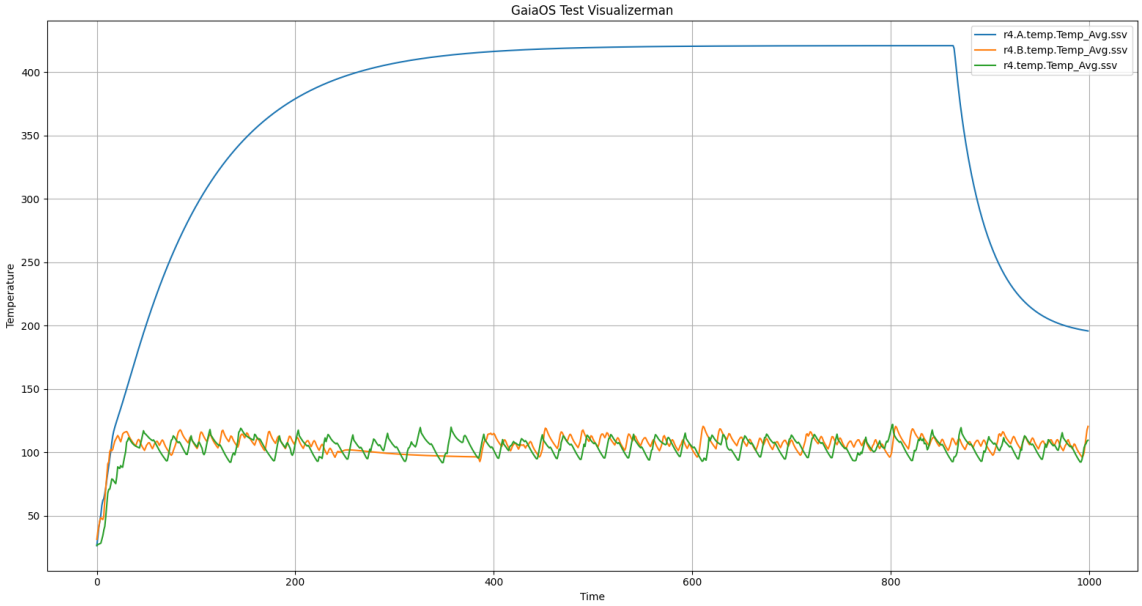


Figure_5

In Figure_5 I've taken the successful r4 and done two more runs of it with a different training seed. This means the training dataset during the random training is different for each one. We see similar results to r4 in the r4.B set, but r4.A gets stuck for most of the test.

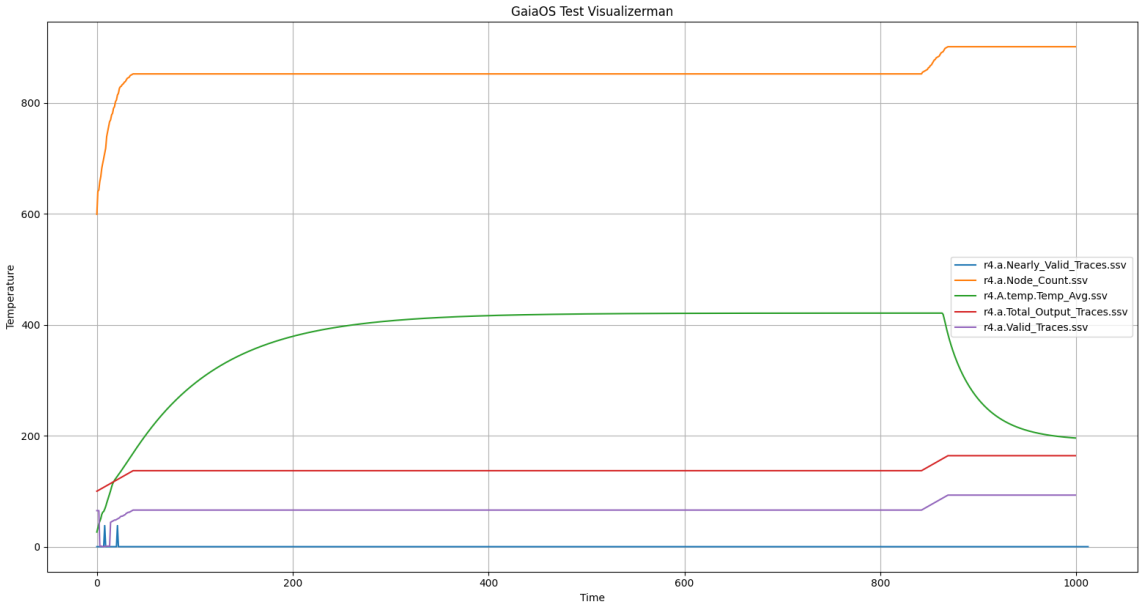


Figure_6

Interestingly something happens at iteration #844 that causes it to retrieve more traces and kicks it into turning on the 22 degree actuator resulting in the dip. Figure_6 shows the upswing in node count, valid traces retrieved, total traces retrieved, and nearly valid traces. To solve this mystery we need to implement more data output from Gaia, we need to see the delta's, the trace output, and the deviation mapping.

My hypothesis is that as the temperatures converged on the actuator temp the rate of change slowed due to the nature of the diffusion simulation, which we can see with r4.A.temp.Temp_Avg.ssv in Figure_6 as it resembles a sigmoid curve.

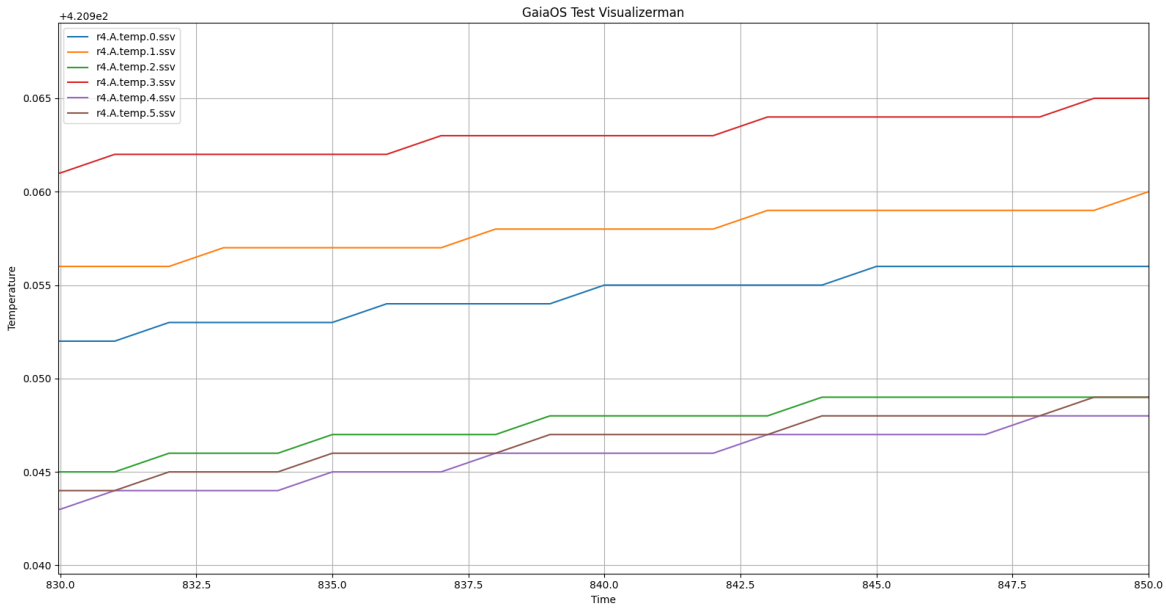


Figure_7

I believe that due to floating point numbers in C++ having a limit to precision the slow diffusion falls below this threshold meaning some sensors report a delta of 0 for several iterations in a row, which is supported by Figure_7.

The system sees these periods of no change as different from the periods of positive delta. With all the temperatures beginning to flip between a delta of (+1) and (0) this results in new memories being encoded as different mixes of the positive and steady delta are encountered. Old memories are triggered by these delta mixes that weren't with the (+1) across the board.

The result of this is that the system recalls memories that cause it to decide to turn on the actuator with a temperature of 22.

Random + Directed Training (c0...c4.D)

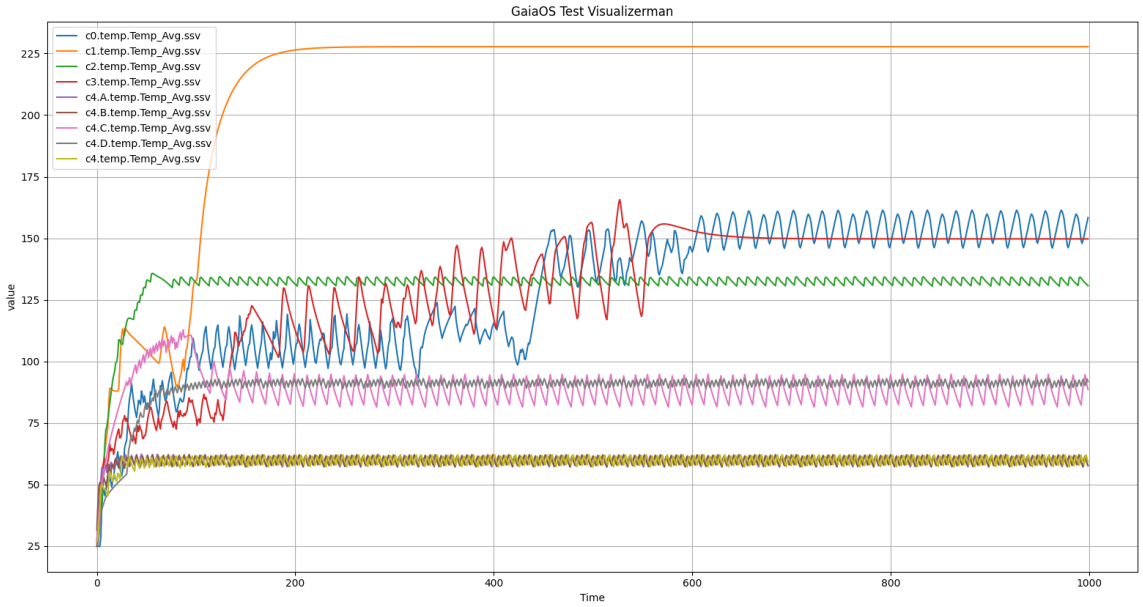


Figure_8

In Figure_8 we can see where only two of the run (keep in mind I've only done one iteration of the lower trained one's in all datasets so the low rate for e & r series could be by chance but it is unlikely imo) failed to 'catch' and stabilize, this is in line with the e-series of experiments and we can infer that there the guided training is likely to thank, as the r-series do not exhibit good rates of catching at lower training levels.

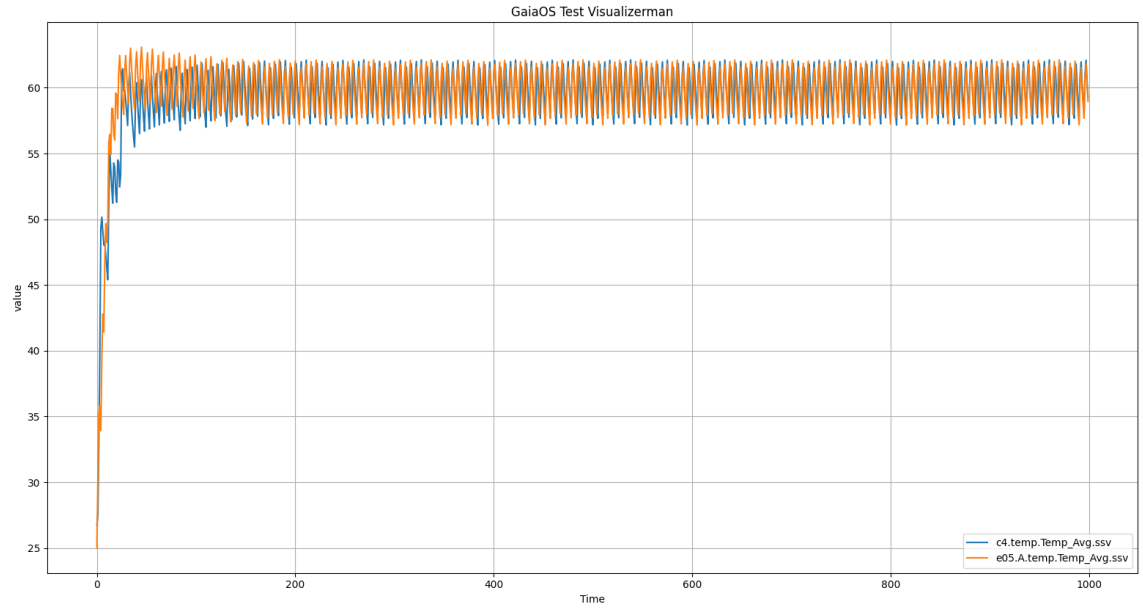


Figure_9

Backing up this theory about the influence of the directed training we can see in Figure_9 the overlay of c4 and e05.A which shows they are nearly identical, if a bit off in timestep and with some variance in the initial exploration.

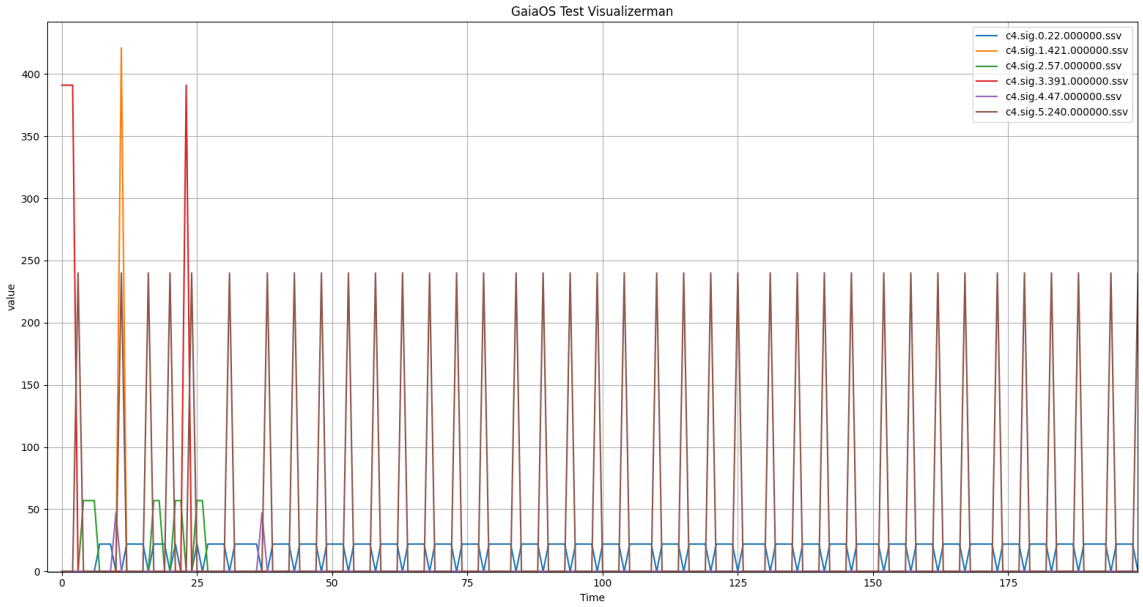


Figure_10

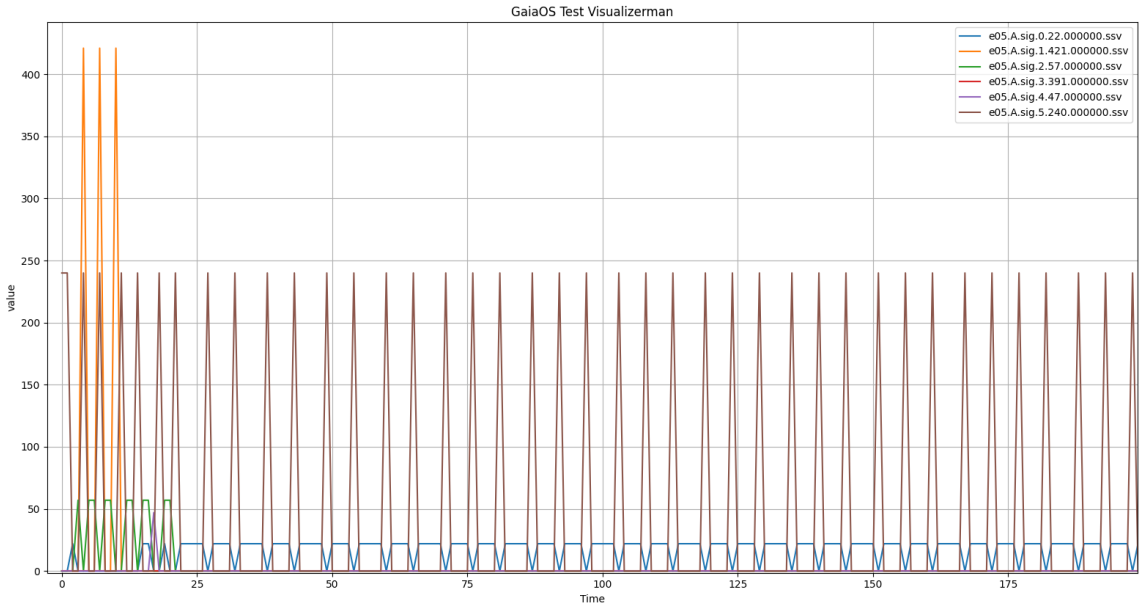


Figure_11

Exploring the signal output of c4 in Figure_10, and e05.A in Figure_11, we see that they have converged upon similar strategies. You can see the initial exploration is different, but the rest is essentially the same.

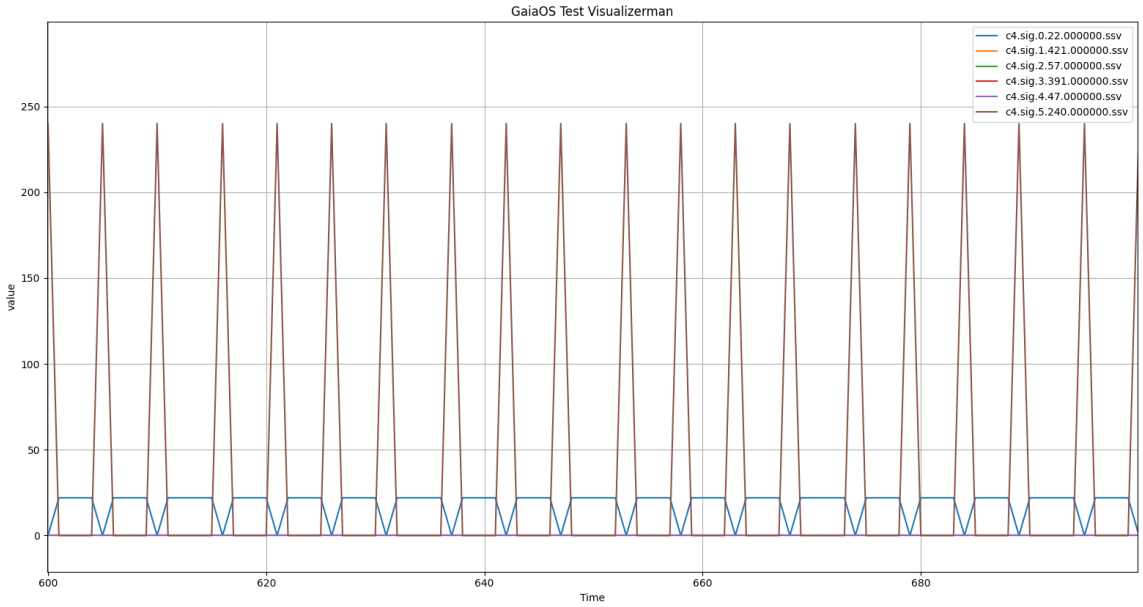


Figure_12

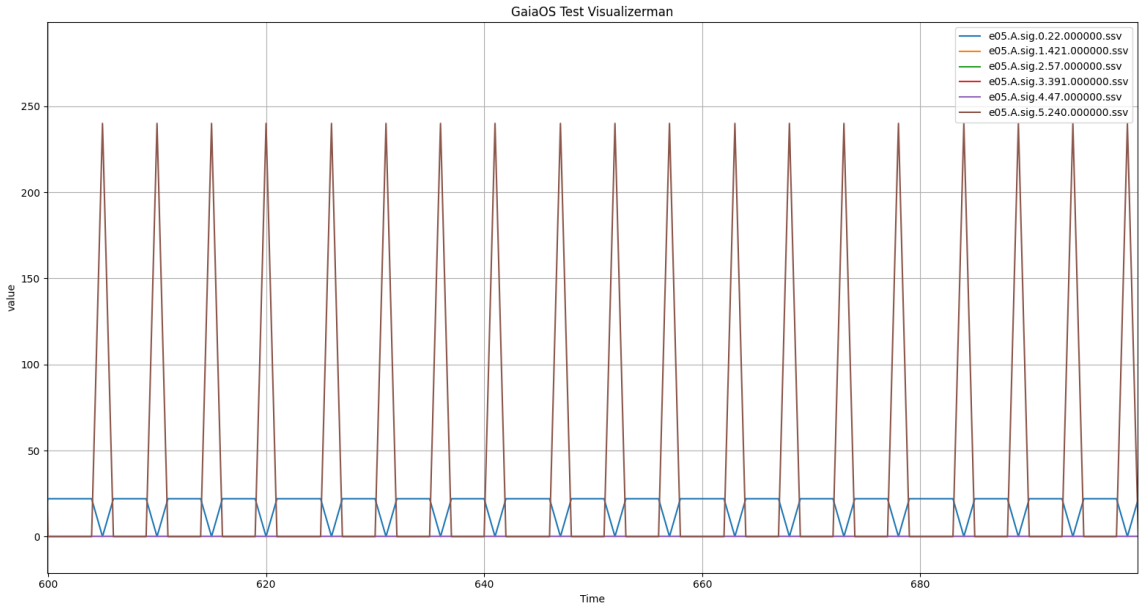


Figure_13

Looking closer with Figure_12 and Figure_13 we can see the similarity in their respective strategies.

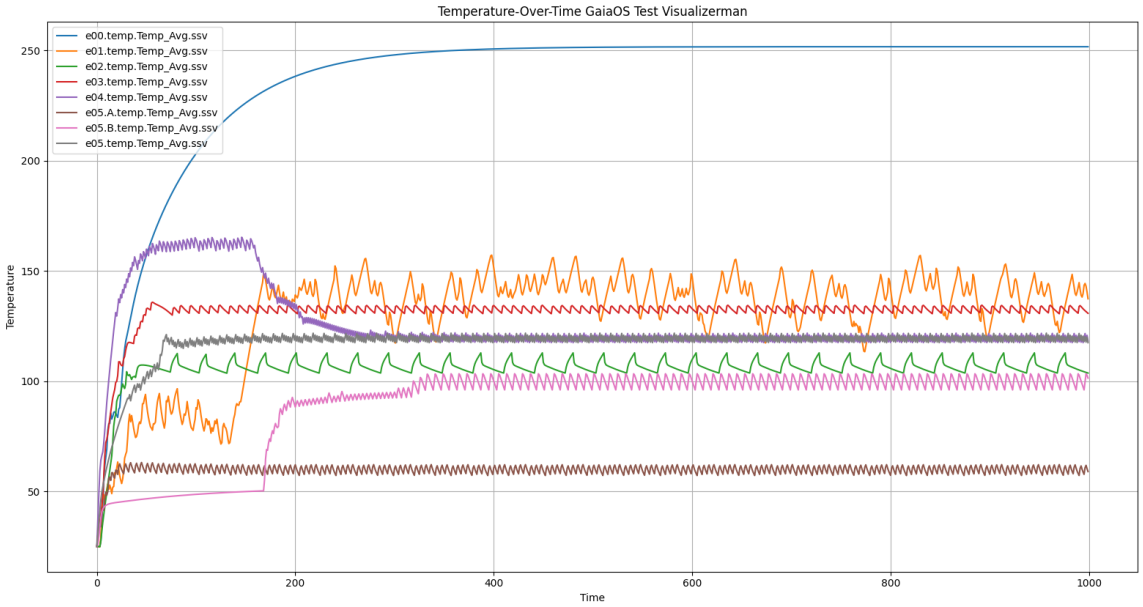


Figure_2

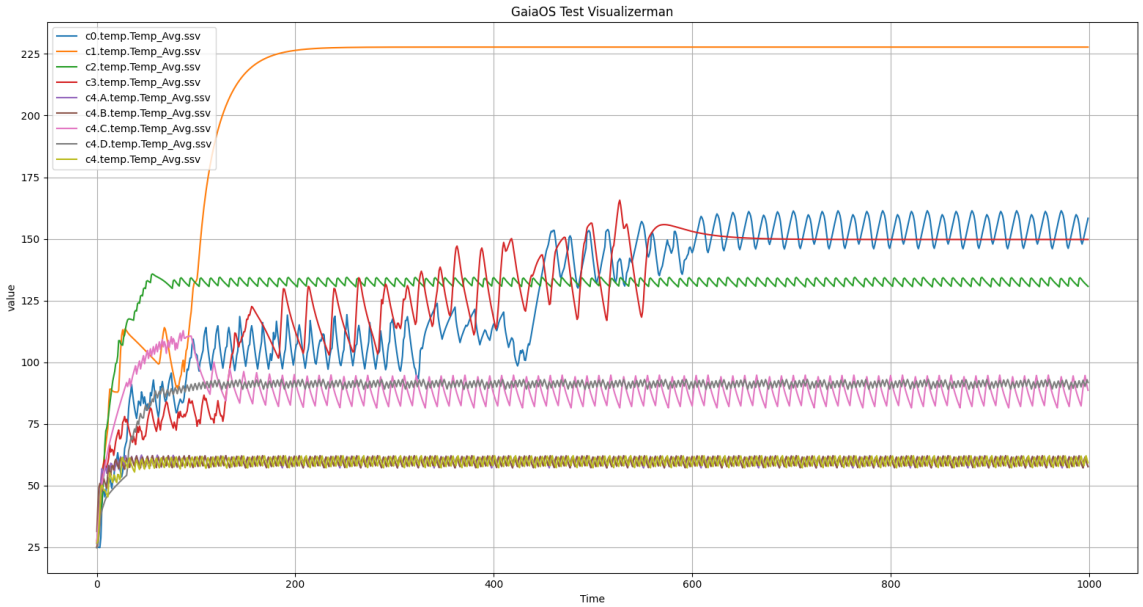


Figure_8

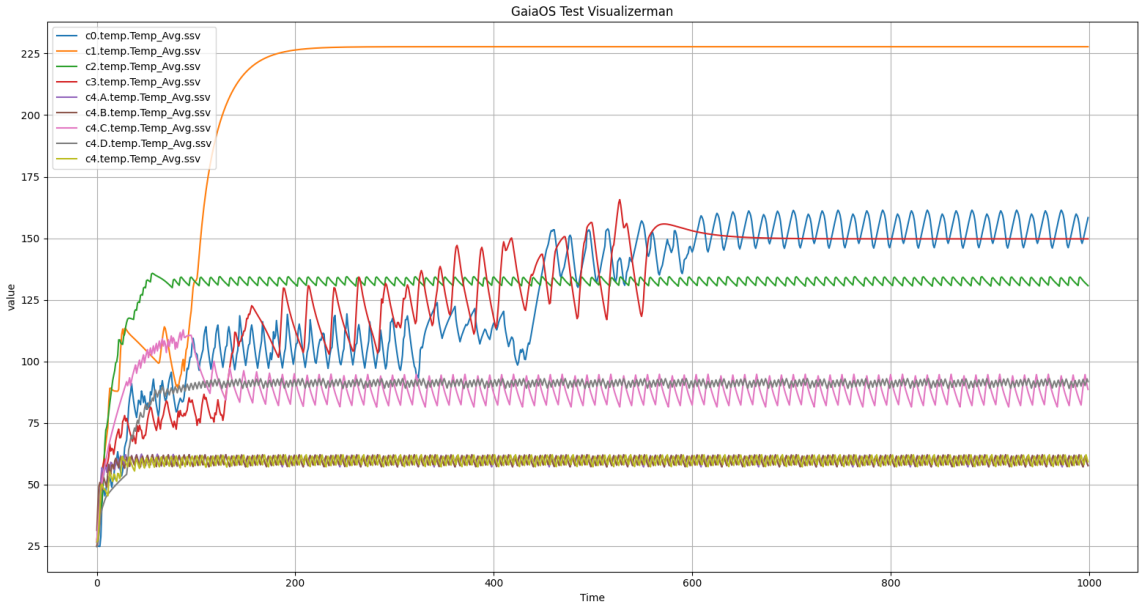


Figure_14

Comparing Figure_2 with Figure_8 the results seem very similar, indicating the strong influence of the directed training. Comparing the c-series in Figure_8 with the r-series in Figure_14 we see a drastic improvement, with most of the r-series becoming stuck.

Conclusions

The key thing is that the system demonstrates the ability to infer controls from the raw data and to apply corrective strategies based on what it has learned in an attempt to control the system.

The system after the initial exploration tends to fall into a cyclical control strategy where it overcompensates. This can be refined through trace selection and the implementation of the prefactor analysis, control relationship exploration during selection, and more.

We see where the system will get into a "rut" and repeat the same strategies over and over getting into a "resonance" with the deterministic diffusion simulation leading to a stagnation in learning.

The nature of the system to get "stuck" is due to the trace selection, it is incorrectly associating signals and not prioritizing previously successful strategies and penalizing errors, causing it to double down on a bad strategy. This will be handled through trace selection refinement, the implementation of prefactor analysis, signal association exploration during selection, implementing success and failure reflection/association, and potentially other methods not yet considered or explored.

From these datasets we can clearly see the benefit of directed exploration, though random exploration shows the ability to learn enough to develop a control strategy it does not seem to be as effective.

Based upon this data the boredom mechanic and exploration mechanic most likely to help the system would be a systematic exploration of the environment, not random flailing.

I dun goofed

I criticize my methodology with the guided training. To set the temperature of the simulation, is that a violation of the goal of unsupervised/self-supervised learning? To manipulate the environment? This realization happened after most of the datasets were generated, so I'm running with it for this project overview due to time constraints but it irks me. Just wanted to put that out there. I've updated the ArcologyX challenge to reflect this, meaning technically Gaia no longer beats Ranran Lvl 1, but that is only a matter of relatively trivial refactoring to fix and a slightly more sophisticated updated exploration algorithm that mimics the random and directed training behavior needed and implements it into their exploration behavior.

Need for more advanced exploration mechanisms

Given what has been learned about the guided and random learning we can infer that a more robust exploratory system would be capable of achieving the same results starting with no pre training at all and immediately trying to control the system as it learns as opposed to an observation stage of training before the test.

Within the modules internal pipeline there will already be a reflection phase so we'll gather information on the current state that we can use during a curiosity/unsure/boredom phase we place before the output but after the reflection stage. Using this gathered information from reflection we can ensure that we don't retry the same thing twice and it can pursue more intentional exploration strategies.

Transformer model adaptation

Drawing from transformer models we can implement a parallel association mechanism alongside the current time-series mechanism.

The system has a "short term memory" in the sense it keeps a shift-register for each I/O in the interface. This can be leveraged to associate the oldest actuator values with the delta values that proceed them in the time-series, the [0] index actuators associated with every sensor delta from [1] to [chrono_depth - 1]. Associating each input with all the changes in the system to come in the next chrono_depth steps.

When a given sensor has a positive delta within the current input space short term memory array it is then associated with each actuator. When a sensor has a negative delta this is associated with every actuator in the [0] index.

Over time the associations to the positive or negative for each sensor will strengthen depending on their frequency. So as the system explores temperatures and finds control strategies is would learn more nuance as the correctly associated deltas overpower the weaker less associated deltas.

Then when we use the time-series to generate predictions we can draw upon the associative system to give another mechanism for the system to infer control and refine strategies. Querying the associative network and taking the aggregate delta outputs will yield another value to incorporate into the final output synthesis.

Sensor deviation attention mechanism:

Drawing from the idea of weighing tokens based on associations within the context window we can apply this idea of weighing important information as heavier to get the network to focus on it.

---=== Appendices ===---

Appendix A:

The diffusion algorithm is shown here as temperature, use whatever you're diffusing following this:

```
<C++>

//Initialization and setup is left out for brevity, this is the core of it, full C++ for the sim can be found on https://github.com/NT4-Wildbranch/proto-Gaia
std::vector<std::vector<std::vector<c_Cell>>> Map;
bool Current_Frame;
bool Next_Frame;

std::vector<c_Sensor> Sensors;
```

```
std::vector<c_Actuator> Actuators;

int Width;

int Height;

bool check_XY(int p_X, int p_Y)
{
    if (p_X < 0) { return 0; }
    if (p_Y < 0) { return 0; }
    if (p_X >= Width) { return 0; }
    if (p_Y >= Height) { return 0; }

    if (Map[Current_Frame][p_X][p_Y].Type == 0) { return 0; }

    return 1;
}

void diffuse_Temp(int p_X, int p_Y)
{
    float tmp_Total = 0.0;
    float tmp_Count = 0.0;
    float tmp_Current = 0.0;

    if (!check_XY(p_X, p_Y)) { return; }

    tmp_Current = Map[Current_Frame][p_X][p_Y].Temp;
    if (check_XY(p_X + 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X + 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X, p_Y + 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y + 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X - 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X - 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X, p_Y - 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y - 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }

    tmp_Current = Map[Current_Frame][p_X][p_Y].Temp;
    if (check_XY(p_X, p_Y - 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y - 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X + 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X + 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X, p_Y + 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y + 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X - 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X - 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }

    tmp_Current = Map[Current_Frame][p_X][p_Y].Temp;
    if (check_XY(p_X - 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X - 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X, p_Y - 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y - 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X + 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X + 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X, p_Y + 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y + 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }

    tmp_Current = Map[Current_Frame][p_X][p_Y].Temp;
    if (check_XY(p_X, p_Y + 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y + 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X - 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X - 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X, p_Y - 1)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X][p_Y - 1].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }
    if (check_XY(p_X + 1, p_Y)) { tmp_Current = (tmp_Current + Map[Current_Frame][p_X + 1][p_Y].Temp) / 2; tmp_Count++; tmp_Total += tmp_Current; }

    Map[Next_Frame][p_X][p_Y].Temp = (tmp_Total /= tmp_Count);
}


for (int cou_X = 0; cou_X < (Width); cou_X++)
{
    for (int cou_Y = 0; cou_Y < (Height); cou_Y++)
    {
        diffuse_Temp(cou_X, cou_Y);
    }
}
</C++>
```

Appendix B:

Parameters to generate the datasets used in this document for yourself

Experiment_Name: e00

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 1

Random Training Depth: 0

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e01

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 5

Random Training Depth: 0

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e02

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 10

Random Training Depth: 0

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e03

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 25

Random Training Depth: 0

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e03.A

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 25

Random Training Depth: 0

random_Run_Seed: 1

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e03.B

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 25

Random Training Depth: 0

random_Run_Seed: 2

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e04

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 50

Random Training Depth: 0

random_Run_Seed: 1

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e05

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 0

random_Run_Seed: 1

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

//This run is actually e05 but I messed up and did e05 with a random run seed of 1 not zero, otherwise that is all that happened here.

Experiment_Name: e05.A

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 0

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: e05.B

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 0

random_Run_Seed: 2

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r0

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 5

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r1

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 10

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r2

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 25

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r3

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 50

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r4

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 100

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r4.A

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 100

random_Run_Seed: 0

random_Training_Seed: 9001

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: r4.B

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 0

Random Training Depth: 100

random_Run_Seed: 0

random_Training_Seed: 9002

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c0

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 5

Random Training Depth: 5

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c1

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 10

Random Training Depth: 10

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c2

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 25

Random Training Depth: 25

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c3

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 50

Random Training Depth: 50

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c4

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 100

random_Run_Seed: 0

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c4.A

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 100

random_Run_Seed: 0

random_Training_Seed: 9001

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c4.B

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 100

random_Run_Seed: 0

random_Training_Seed: 9002

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c4.C

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 100

random_Run_Seed: 1

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0

Experiment_Name: c4.D

Chrono_Depth: 4

Step_Count: 25

Directed Training Depth: 100

Random Training Depth: 100

random_Run_Seed: 2

random_Training_Seed: 9000

Prediction APT MSC: 0.99

Prediction MC MSC: 0.95

Prediction APT Chrono: 0.99

Prediction MC Chrono: 0.95

Deviation APT MSC: 0.0

Deviation MC MSC: 0.95

Deviation APT Chrono: 0.0

Deviation MC Chrono: 0.95

Bulk_Gen: 0