C++ Based Neural Network Simulation Environment Design & Implementation

Finian J. Graham – 02/04/2024

Contents

[Design 3](#_Toc163995492)

[Simulation Environment 3](#_Toc163995493)

[Overview 3](#_Toc163995494)

[Tools to Utilize 3](#_Toc163995495)

[Environment Design 3](#_Toc163995496)

[Network 4](#_Toc163995497)

[Overview 4](#_Toc163995498)

[Network Components 4](#_Toc163995499)

[Systems Overview 5](#_Toc163995500)

[Actions and Sensors 5](#_Toc163995501)

[Learning and Evolution Algorithm 5](#_Toc163995502)

[Overview 5](#_Toc163995503)

[Learning Algorithm Design 5](#_Toc163995504)

[Implementation 7](#_Toc163995505)

[System 7](#_Toc163995506)

[UI 8](#_Toc163995507)

[Dashboard 8](#_Toc163995508)

[World State View 8](#_Toc163995509)

[Grid Implementation 9](#_Toc163995510)

[Overview 9](#_Toc163995511)

[Grid Update Flow 9](#_Toc163995512)

[Agent Implementation 10](#_Toc163995513)

[Overview 10](#_Toc163995514)

[Reward Function 10](#_Toc163995515)

[Survival 10](#_Toc163995516)

[Mutation 10](#_Toc163995517)

[Agent Update Flow 11](#_Toc163995518)

[Brain Implementation 12](#_Toc163995519)

[Overview 12](#_Toc163995520)

[Actions and Sensors 12](#_Toc163995521)

[Network 12](#_Toc163995522)

[Brain Update Flow 13](#_Toc163995523)

[Learning Algorithm Implementation 14](#_Toc163995524)

[Overview 14](#_Toc163995525)

[References 15](#_Toc163995526)

# Design

## Simulation Environment

### Overview

The simulation environment will be used to test and evaluate the effectiveness of any implemented learning and evolutionary algorithms. It will include a dashboard with simulation controls and graphing tools in order to visualize how the simulation is performing and how each neural network has developed, and a simulation view that will display the current world state.

### Tools to Utilize

For User Interface implementation I will be using the ImGui library due to its simplicity and efficiency in rendering, allowing for the creation of data visualization and controls. (Cornut, 2014) In addition to the base ImGui library, I will be using the ImPlot extension (Pezent, 2020) that allows the creation of simple graphs in order to display the data required.

For rendering the simulation, I will be using OpenGL (Khronos Group, 2021) in order to utilize simple shaders to render the world state. ImGui is also compatible with OpenGL, so the two libraries will work well together.

For saving and loading data the Cista serialization library will be used. This is a single header library that enables simple saving and loading into files and will be used to handle the storing and retrieving of brain data for testing. (Guendling, 2023)

### Environment Design

To increase flexibility and space to display data, the Dashboard and the World State will be separated into two different resizable windows. All controls and data will be stored on the Dashboard, with the user being able to interact with the world view by clicking on cells and manipulating the view by zooming in and out with the scroll wheel and moving the camera by using the arrow keys. Figure 1 Is a proposed layout design for the dashboard, with the following sections labelled:

A – General simulation controls.

B – Neural network visualization and selected agent data.

C – Selected graph plot

D – General simulation data such as number of agents, oldest agent, highest energy gain, etc…

A diagram of a network

Description automatically generated

Figure 1: Dashboard Layout Design

## Network

### Overview

The network will be designed from the ground up in C++ to allow for flexibility and experimentation with different mutations and unconventional architectures. Building everything in C++ also allows the system to be implemented in a wide range of projects, due to C++ being one of the most popular programming languages found in games today.

### Network Components

The network will be comprised of three types of components. Layers, Links and Neurons will be utilized to create the network architecture required for machine learning.

**Layers** are comprised of multiple Neurons, ensuring that the order of the network is known and preventing cyclical links. Neurons within a layer can only form links with neurons in the next layer along.

**Links** contain an input neuron, an output Neuron, and a weight. This weight can be adjusted to optimize the overall network during learning.

**Neurons** contain a Bias and an Activation value that determines what value is passed along the output links during forward propagation. This activation value differs based on what kind of neuron is being used, with input neurons taking a value from the world state, and hidden and output neurons taking the weighted sum of all input links. (Amazon, 2024)

For my implementation there will be different Neuron types that handle incoming and outgoing signals differently. The most basic types will be Sensor, Hidden and Action neurons.

* Sensor neurons base their activation on the world state and output this value to the output links.
* Hidden neurons take the input link values and calculate the weighted sum before outputting this value to the output links.
* Actions neurons take the input link values and then calculate the probability of that action being triggered based on other action neuron values

### Systems Overview

### Actions and Sensors

Actions and sensors act as the bridge between the brain and the agents. The aim is to minimize the direct reliance of the brain on the agent’s data, and have any data manipulation occur via Actions and Sensors. When the network wants the agent to do something in simulation space, it should rely on an Action to directly influence the Agent, and when the network needs information from the Agent, it should rely on a Sensor.

My proposed implementation of this system is to create a global instance of an “Actions” and “Inputs” class that contains static functions for each action or input required. These functions can then be referenced via function pointers by the network’s neurons, with data being passed in via an InputData or ActionData struct.

Since the above approach will prevent storing data within the Action or Input class, an std::variant (cppreference, 2024) will be utilized to ensure that each function signature contains the same format, while still allowing different formats of data to be passed to the functions. This data can then be stored in the respective Neuron for future use.

## Learning and Evolution Algorithm

### Overview

The implementation of the learning and evolution algorithms will aim to combine Deep Reinforcement Learning and Survival of the Fittest theory in order to create a stable yet evolving ecosystem populated with agents that can survive and interact with the world state.

### Learning Algorithm Design

The project will implement a Policy-Based algorithm called **Reinforce**. (Hugging Face, 2024) Reinforce is an algorithm that aims to increase the probability of actions based on how successful the actions are, while using Monte-Carlo sampling in order to generate a series of actions that will be used to train the network. Along with this, a discounted reward formula will be used in order to determine a more accurate value for each action. Figure 3 shows the proposed discounted reward formula, in which the reward is the sum of all future action rewards, discounted by the depth value. (Thomas Simonini, 2022)A math equations and formulas

Description automatically generated with medium confidence

Figure 2: Discounted Reward Formula

This reward formula will then be used to determine the increase or decrease in weights for each action checked during the sequence, in order to increase the probability of good actions occurring and decreasing the probability of bad actions. Figure 4 displays the proposed formula to use for Reinforce learning. (Hugging Face, 2024)

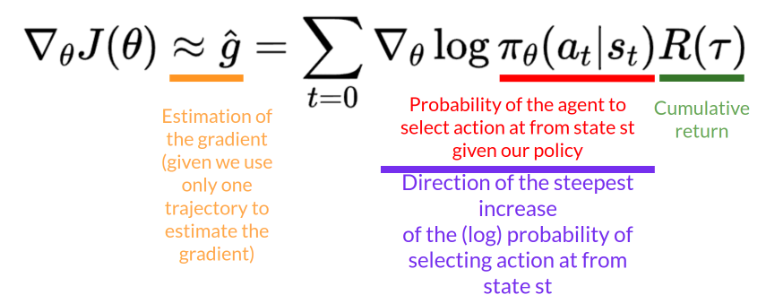


Figure 3: Reinforce Policy Gradient Formula (Hugging Face, 2024)

Effectively, since we cannot check every single outcome for the best reduction in our Objective value J(θ) without extreme computational costs, the aim is to estimate the gradient using the direction of the steepest increase in probability and the cumulative reward.

An additional step that may be tested is introducing multiple episodes. This would theoretically increase the effectiveness of the algorithm, but would increase the computational requirements drastically. Should multiple episodes be tested, the following formula highlighted in Figure 5 would be implemented. (Hugging Face, 2024)

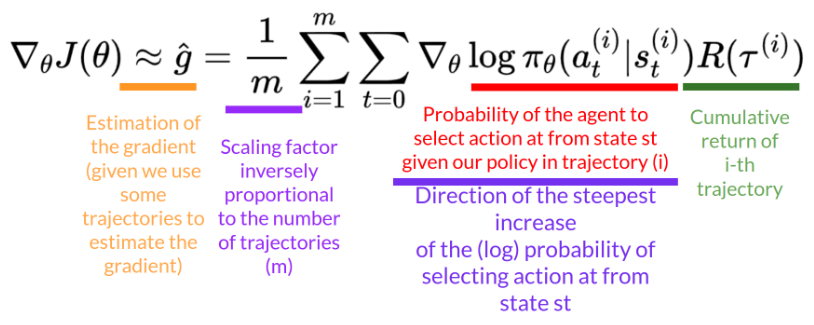


Figure 4: Policy Gradient Formula Including Multiple Episodes (Hugging Face, 2024)

To prevent the ecosystem from stagnating completely, a Survival of the Fittest style neuro-evolution system will be added. After each successful reproduction, the new child agent will have a chance to mutate. Mutations that make agents better than their parents will live on, and theoretically the ecosystem should eventually contain multiple evolutionary paths that create different survival strategies.

# Implementation

## System

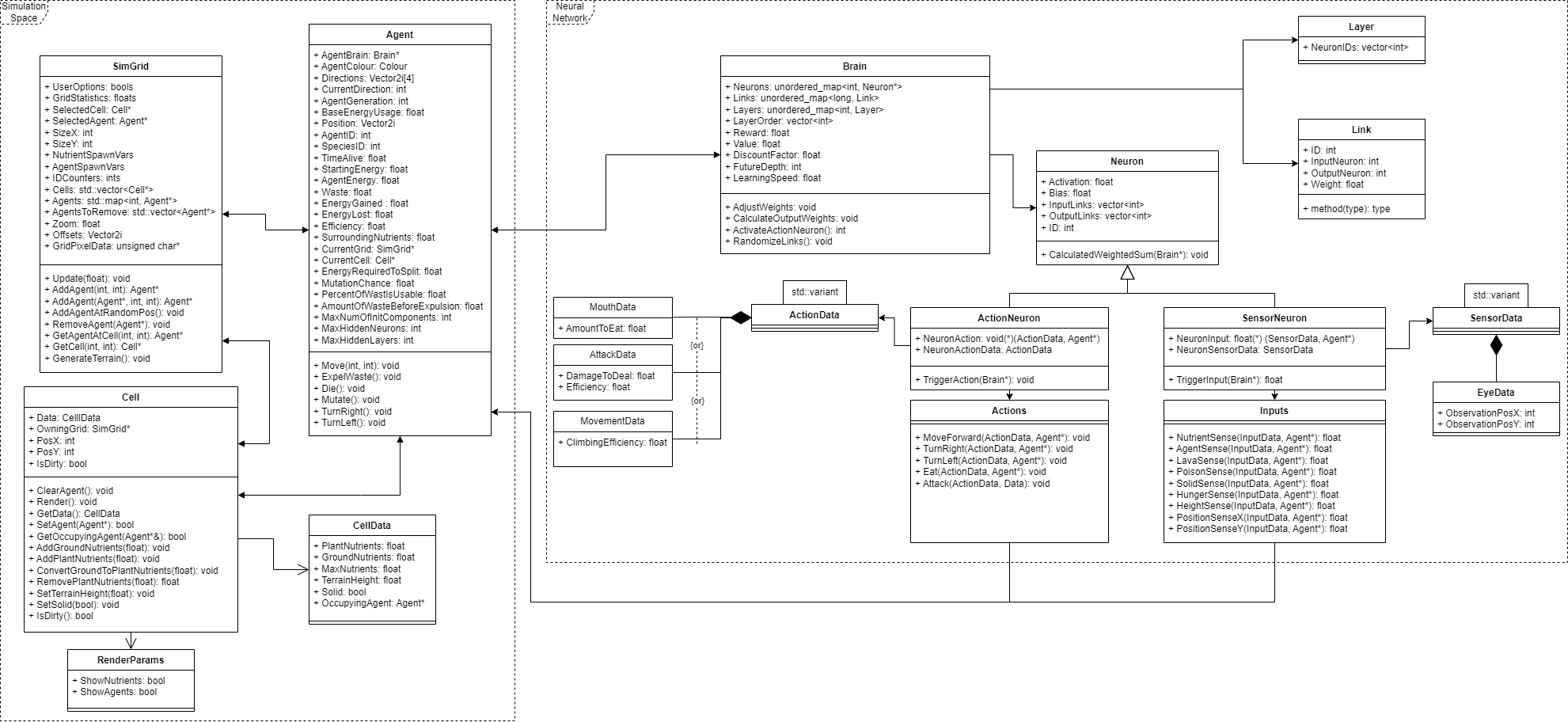


Figure 5: System Class Diagram

## UI

### Dashboard

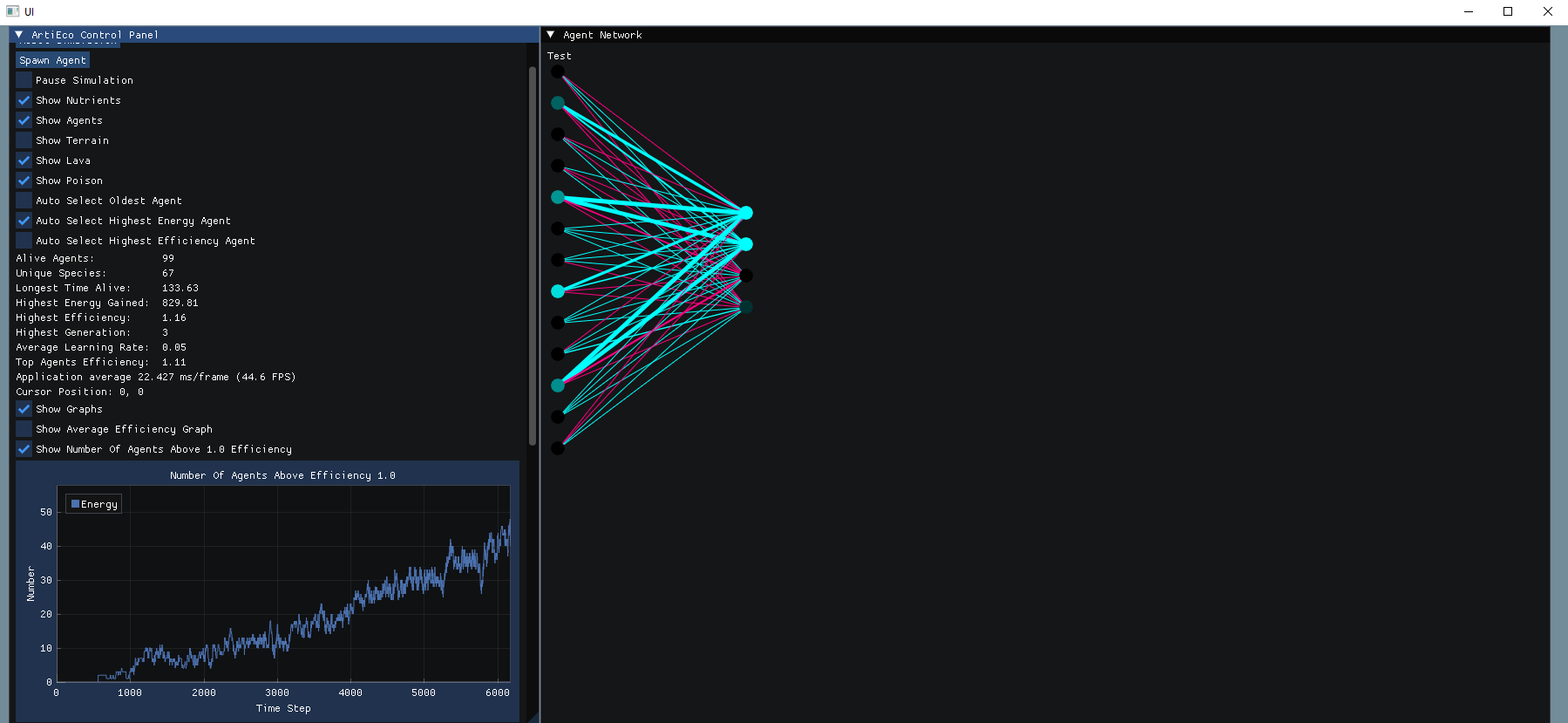


Figure 6: Dashboard Visual

### World State View

A green and purple pixelated background

Description automatically generated

Figure 7: Grid Visual

## Grid Implementation

### Overview

The grid is made up of cells that contain data required to simulate the ecosystem. Each cell contains the following data:

* How many Plant Nutrients are present.
* How many Ground Nutrients are present.
* The current terrain height.
* If the cell is solid.
* If the cell is poisoned.
* If the cell is lava.
* What agent is currently occupying the cell.

Each cell can only be occupied by one agent.

### Grid Update Flow

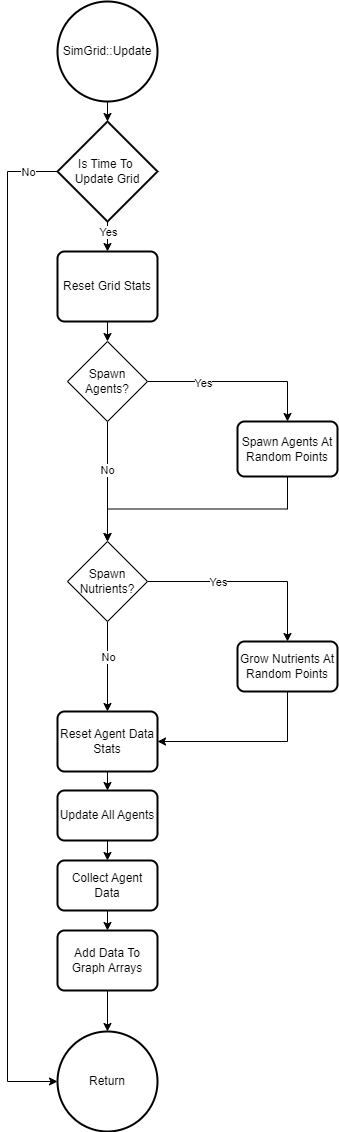


Figure 8: Grid Update() function flow chart

## Agent Implementation

### Overview

The Agent acts as the brain’s connection to the World State. On spawn, an agent randomly generates a selection of inputs and outputs for the brain to utilize.

### Reward Function

The Agent determines how good the previous action it took was based on the agent’s state before and after the action is taken. The following values are considered:

* How much energy is lost or gained.
* How much health is lost or gained.
* How many more nutrients are close to the agent.
* How much more lava is close to the agent.
* How much more poison is close to the agent.

All of these values are modified by the agent’s sensitivity for the different aspects, with higher sensitivities causing the variable to have a greater effect on the overall reward value. These sensitivities are randomly chosen when the agent is first spawned and can mutate to become more or less sensitive on reproduction.

### Survival

Each agent is given two vital variables that control if they live or die.

Energy is a resource which is gained or lost based on how much the agent eats, and how many actions the agent takes. If an agent gains enough energy, the agent will reproduce and a copy of the agent will be created, with the possibility of mutation.

Health determines how damaged an agent is, and certain hazards will reduce the agent’s health. Agents slowly regain health over time.

If either energy or health runs out, the agent dies and is removed from the world state.

### Mutation

After an agent reproduces, there is a chance that the child agent will contain a mutation. This mutation can affect a wide range of variables within the child and aim to both increase the chance of good agents being produced and increasing the diversity of the ecosystem. Each agent contains a “stability” variable, which determines the likelihood of children mutating.

Currently the mutations that can occur are:

* Removing or adding a sensor
* Removing or adding an action
* Removing or adding a link
* Removing or adding a hidden neuron
* Increasing or decreasing a sensitivity
* Increasing or decreasing the agent’s stability

### Agent Update Flow

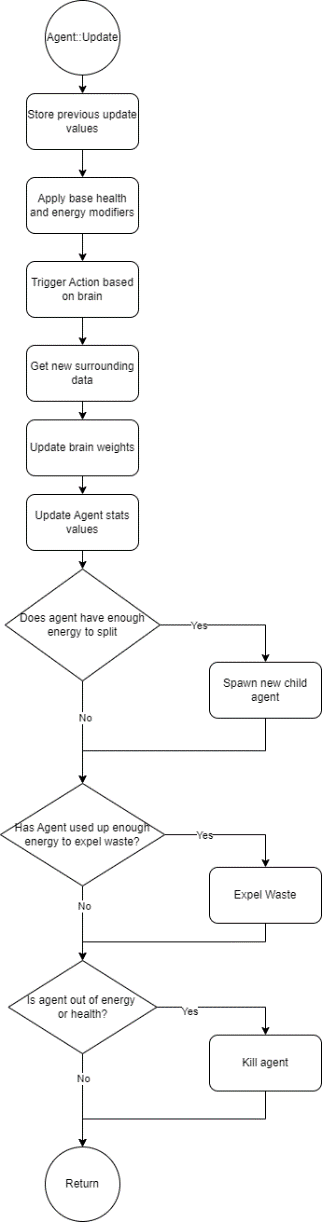


Figure 9: Agent Update Flow Chart

## Brain Implementation

### Overview

The brain is implemented in a way so that future adjustments to the learning algorithm are easy to implement. After each action is carried out by the agent, UpdateWeights is called which uses the implemented algorithm to adjust the network’s weights.

### Actions and Sensors

To control the agent and receive information from the agent for network processing, a series of Action and Sensor functions have been created. These functions are stored in a global Actions and Inputs object, and then referenced by the required Action and Sensor neurons respectively. The only data that the brain gets directly from the agent is the Reward value, and everything else is processed by the Sensors.

Each Action and Sensor function is provided an Agent pointer, and a Data variant structure. This Data structure is used to pass unique values to the functions related to what the neuron requires.

### Network

The Network consists of the following structures:

|  |  |
| --- | --- |
| **Layer** | |
| Contains a list of Neuron IDs. The brain contains an array of Layers that dictates the order. | |
| **Variable Name** | **Use** |
| std::vector<int> NeuronIDs | Contains all neuron IDs within the layer. |

|  |  |
| --- | --- |
| **Link** | |
| Links determine which neurons are linked together. Each link is identifiable via an ID, which is defined by a pairing function ((a + b) \* (a + b + 1)) / 2 + b (Pigeon, 2024) where a is the input neuron’s ID and b is the output neuron’s ID. This method of ID generation works because neurons can only be linked in one direction. | |
| **Variable Name** | **Use** |
| int ID | Link’s unique ID within the brain. |
| int InputNeuron | Input neuron ID. |
| int OutputNeuron | Output neuron ID. |
| float Weight | The link’s current weight in the network. |

|  |  |
| --- | --- |
| **Neuron** | |
| Neurons process input values and output a new value based on the weighted sum of all input strengths. Sensor Neurons base their activation on the world state, while action neurons do not have outputs. Instead, the next action is based on a probability driven by which Action Neuron has the highest activation. | |
| **Variable Name** | **Use** |
| float Activation | The neuron’s current activation value. |
| float Bias | The bias added to any inputs. |
| std::vector<int> InputLinks | The neuron’s input links. |
| std::vector<int> OutputLinks | The neuron’s output links. |
| int ContainingLayer | The layer ID containing this neuron. |
| int ID | The neuron’s unique ID within the brain. |

### Brain Update Flow

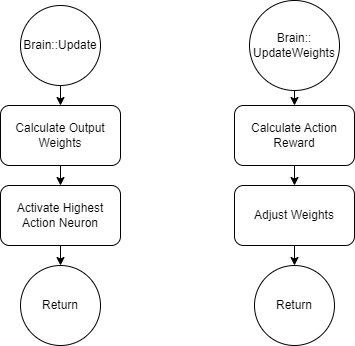


Figure 10: Brain Update and UpdateWeights functions

## Learning Algorithm Implementation

### Overview

For testing, two different algorithms are currently in place. The simplified algorithm takes the concept of REINFORCE and only applies weight shifts to the links directly connected to the initial action taken. The full algorithm goes through each action taken during simulation and applies the weight shifts to each one, compensating for simulation depth.

Simplified algorithm pseudo code:

|  |
| --- |
| initialize RewardValue = 0.0  initialize DiscountFactor  RewardValue += Current Action Reward  **repeat** until I == FutureDepth + 1  simulate future action  RewardValue += simulated action reward \* DiscountFactorᶦ  **end repeat**  **for each** Link to neuron  L = ∇log(Action Probability)  Gradient = L \* RewardValue  Link Weight -= Gradient  **end for** |

Full algorithm pseudo code:

|  |
| --- |
| initialize RewardValue = 0.0  initialize ActivatedNeurons vector  initialize ActionRewards vector  initialize ProbabilityValues vector  **repeat** **until** I == FutureDepth + 1  simulate future action  ActivatedNeurons[I] = LastActivatedNeuron  ProbabilityValues[I] = Action Probability  ActionRewards[I] = simulated action reward \* DiscountFactorᶦ  **end repeat**  **for** I **in** ActionRewards  ActionRewards[I] = discounted reward of future actions  **end for**  **for each** Neuron **in** ActivatedNeurons  ActionDepth = depth of action in simulation  **for each** Link to neuron  L = ∇log(ProbabiltyValues[ActionDepth])  Gradient = L \* ActionRewards[ActionDepth]  Link Weight -= Gradient  **end for**  **end for** |

# References

Amazon. (2024, April 02). *What is a Neural Network*. Retrieved from aws: https://aws.amazon.com/what-is/neural-network/

Cornut, O. (2014, August 11). *Dear ImGui*. Retrieved from Github: https://github.com/ocornut/imgui

cppreference. (2024, April 14). *std::variant.* Retrieved from cppreference: https://en.cppreference.com/w/cpp/utility/variant

Guendling, F. (2023, June 02). *Simple C++ Serialization & Reflection*. Retrieved from Github: https://github.com/felixguendling/cista

Hugging Face. (2024, April 14). *Diving deeper into policy-gradient methods.* Retrieved from Hugging Face: https://huggingface.co/learn/deep-rl-course/en/unit4/policy-gradient

Hugging Face. (2024, April 04). *The Problem of Variance in Reinforce*. Retrieved from Hugging Face: https://huggingface.co/learn/deep-rl-course/en/unit6/variance-problem

Khronos Group. (2021, June 04). Retrieved from OpenGL: https://www.opengl.org/

Pezent, E. (2020, May 11). *ImPlot*. Retrieved from Github: https://github.com/epezent/implot

Pigeon, S. (2024, April 10). *Wolfram MathWorld*. Retrieved from Pairing Function: https://mathworld.wolfram.com/PairingFunction.html

Thomas Simonini, O. S. (2022, May 04). *An Introduction to Deep Reinforcement Learning*. Retrieved from Hugging Face: https://huggingface.co/blog/deep-rl-intro