Extending a Neural Simulator to Combine Growth and Spike-Timing-Dependent Plasticity

Yi-Hsin Emily Hsu

A report

submitted in partial fulfillment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2020

Project Committee:

Michael Stiber, Chair

Munehiro Fukuda

Wooyoung Kim

Table of Contents

1. Abstract

Neural network development passes through phases, which include growing the network and its connections to tuning those connections. One of the major mechanisms of that tuning process is called spike-timing-dependent plasticity (STDP). In STDP, the strength of a synapse — the connections between two neurons — is influenced by spike order and timing in those neurons. Understanding the effect of STDP on neural network development is a central question in neuroscience. In this project, we built the infrastructure to help answer this question by extending an existing neural simulator to enable simulations that combine network growth and STDP tuning. We first implemented a serialization and deserialization capability, so that the simulation state information from growth could be used as input for STDP simulation. We also modified an existing STDP synapse class following a mathematical model from literature. The project ended with a large-scale network growth simulation followed by the STDP tuning. The tuning results demonstrated synaptic weight distribution was shifted from unimodal to bimodal.

2. Introduction

The development of neural networks is composed of two major phases: growing a network and tuning the network. At the growing stage, neurons transmit signals and synapses (the connections) are formed. Once the connections have been established, the tuning process begins for connection adjustment. In this tuning phase, one of the major mechanisms is called spike-timing-dependent plasticity (STDP). In STDP, a synapse is strengthened when presynaptic spikes precede postsynaptic spikes. It is weakened when postsynaptic spikes precede presynaptic spikes. The order and timing of spikes determine if synapses are strengthened or weakened, causing an increase or a decrease in synaptic transmission.

Understanding the effect of STDP on neural network development is a central question in neuroscience. In this project, we extend an existing neural simulator to enable simulations that combine network growth and STDP tuning. We first implement a serialization and deserialization feature to the simulator, so that essential simulation states from growth can be serialized and serve as input for STDP simulation. We also modify an existing STDP synapse class by following a mathematical model from previous literature to adjust synapses based on spike timing. The project ends with a successful demonstration of extending a growth simulation with a STDP simulation.

This project is part of the BrainGrid project, under the direction of Dr. Michael Stiber at the University of Washington Bothell. The goal of this project is to enable simulations of network growth and STDP tuning by 1) implementing a serialization and deserialization feature and 2) modifying an existing STDP synapse class. This project hopefully can provide a computational solution in understanding STDP in neural network development.

3. Background: Biological Neural Networks

3.1. Neurons, Synapses, and Neural Networks

Biological neural networks are complex systems which coordinate behaviors in response to stimuli. The primary functional unit in neural networks is called a *neuron*, or *nerve cell*. Its main functionality is signal transmission. A neuron has three defined regions (Figure 1): *cell body* (or *soma*), *axon*, and *dendrites*. The cell body includes the neuron's nucleus, which has the genes of the neuron. Extended from the cell body is a cylindrical structure called the axon. The axon is responsible for sending signals to other neurons. There are also branch-like structures extended from the cell body called dendrites [1]. Dendrites are used to receive signals from other neurons [2]. Axon and dendrites are also called *nerve fibers* or *neurites*. A bundle of nerve fibers is commonly known as *nerve*.

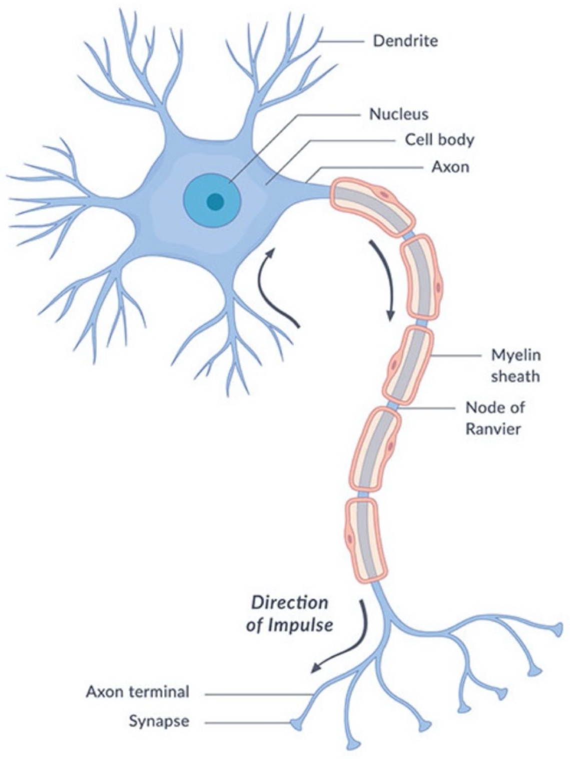


Figure 1: The structure of a neuron [3]. The main parts are the cell body, axon, and dendrites. The cell body includes the neuron's nucleus, the axon is a cylindrical structure for sending signals, and dendrites are branch-like structures for receiving signals.

Another critical component of the neural network is the *synapse*. The synapse is essentially the connection point between two neurons. It is important because it controls the communication among neurons in the network. When a neuron sends signals to another neuron, these signals are passed from the axon of a neuron through the synapse and finally delivered to the dendrites of another neuron. Typically, the neuron that sends signals to the synapse is referred to as a *presynaptic neuron* or *source neuron*. The neuron that receives signals from the synapse is called a *postsynaptic neuron* or *destination neuron*.

3.2. Spikes

The signals transmitted among neurons are called *spikes* or *action potentials*. Specifically, a spike is an electrical activity describing a significant change of a neuron's *membrane potential*, an electrical potential difference between the inside and outside of the cell. The membrane potential is normally about -65 mV [1] at rest. If there is an *excitatory* input signal, the membrane potential will increase, causing the neuron to be more likely to spike. If there is an *inhibitory* input signal, membrane potential will decrease, causing the neuron to be less likely to spike. A spike only happens when the potential reaches a threshold. That is to say, an increase in potential doesn't guarantee a spike; however, if it passes the threshold, a full spike will be generated. This mechanism is called the *all-or-none law*. After the spike, the membrane potential will instantly drop to a point that is slightly lower than resting potential and gradually recover to the resting state, which is generally known as the *refractory period*.

In fact, since spikes are often thought of as all-or-none, a single spike doesn't carry any information. Instead, the signals are believed to be encoded with the frequency and timing of spikes. Therefore, previous studies mainly focus on the temporal or spatial spiking activities. These activities can sometimes lead to network behavior. For instance, a *burst* is a network behavior involving rapid spiking activities and even synchronized spiking among most or all neurons in the network [4].

3.3. Spike-Timing-Dependent Plasticity

Spike-timing-dependent plasticity (*STDP*) refers to the adjustment of the strength of the synapse based on the order and relative spike timing of two connected neurons. In neural networks, the strength of synapse, termed *synaptic weight,* is the amount of impact a neuron has on another when producing a spike.

STDP has been widely discussed in the last two decades and serves as a central learning rule utilized in many computation models [5]. The most influential theory about STDP is called *Hebb's postulate*. The central idea of Hebb's postulate is described as: when a neuron consistently contributes to the spiking of another neuron, the strength of synapse is increased. In fact, in earlier plasticity studies, most focus on the causes of plasticity to the frequency of neurons, not spike timing [6]. In 1997, Markram et al. studied spike timing on plasticity and discovered that the order and timing of presynaptic and postsynaptic spikes indeed changed the extent of synapses, beginning the era of STDP [7].

3.3.1. Principle of Spike-Timing-Dependent Plasticity

The basic principle of STDP states that when presynaptic spikes precede postsynaptic spikes by ~0 to 20 ms, the synapse strength is strengthened, known as *long-term potentiation (LTP)*. Whereas when postsynaptic spikes precede presynaptic spikes by ~0 to 20-100ms, the synapse strength is weakened. This is known as *long-term depression (LTD)* [5], [8]–[11]. The synapse being strengthened or weakened causes an increase or a decrease in synaptic transmission, respectively [12].

3.3.2. Mathematical Models of STDP

In this project, we used the STDP mathematical model presented in [8]. The model describes the contribution of each pre-post spike pair to synaptic modification is formalized as:

$$\Delta W_{potentiation} = \mathrm{A}_{+} e^{-|\Delta t|/\tau_{+}} \quad if\ \Delta t > 0 \tag{3.1}$$

$$\Delta W_{depression} = \mathrm{A}_{-} e^{-|\Delta t|/\tau_{-}} \quad if\ \Delta t < 0 \tag{3.2}$$

$$\Delta t = t_j^{post} - t_i^{pre} \tag{3.3}$$

ΔW is the fractional change in synaptic weight, A is the scaling factor, τ is the time constant, $+$ and $-$ represent potentiation and depression, respectively. Δt is the postsynaptic spike time minus the presynaptic spike time, j and i mean the jth spike in postsynaptic neuron and the ith spike in presynaptic neuron. Figure 2 shows the plot of the mathematical model.

To combine each spike pair's contribution in weight adjustment, the multiplicative model [8] was used. The multiplicative model is formalized as:

$$1 + \Delta W = \prod_{ij} (1 + \Delta W_{ij}) \tag{3.4}$$

where ΔW_{ij} is the contribution of each spike pair.

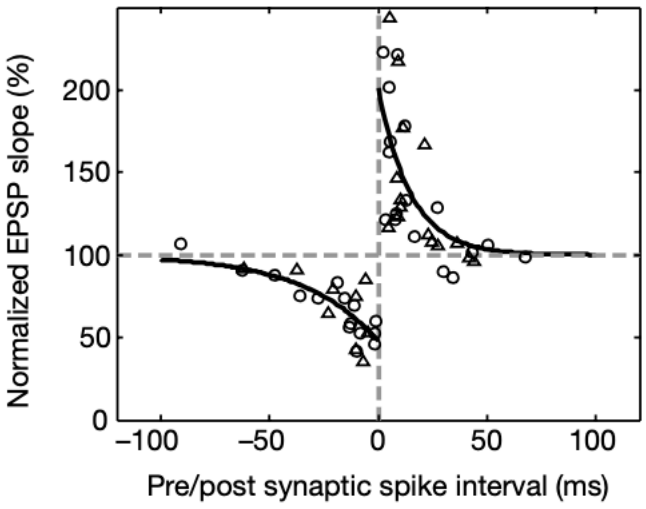


Figure 2: The STDP model [8]. The x-axis is the spike interval defined as the postsynaptic spike time minus the presynaptic spike time. The y-axis is the fractional change in excitatory postsynaptic potential (EPSP), a measure of the membrane potential in the postsynaptic neuron. Each point represents one experiment's data. The drawing curves are the math model defined in (3.1), (3.2), and (3.3).

4. Background: BrainGrid Simulator

4.1. BrainGrid Simulation

BrainGrid [13] is a complex C++ program for neural network simulations. It includes many computational models to simulate various network phenomena. By far, BrainGrid has more than 50 classes and enables CPU-based, GPU-based, single-threaded or multi-threaded neural network simulations. In BrainGrid, simulations capture individual neuron activities as well as whole network behaviors. The data generated provides scientists with a good amount of resources in researching the underlying mechanism of neural networks.

4.1.1. BrainGrid Program Structure

The classes in BrainGrid are organized into different subsystems. For instance, there is a *Core* subsystem that has all the key classes driving the simulation. The *Neuron* and *Synapse* subsystems involve neural and synaptic models used in computation of simulations. Subsystems like *Recorders* and *Utils* are basically tools utilized in simulations.

Figure 3 presents several key classes in BrainGrid. Two major classes, the SimulationInfo class and the Simulator class, are included in the Core subsystem. The SimulationInfo class stores network data, whereas the Simulator class has operation methods. In the SimulationInfo class, there is a class named Model, which includes four foundation classes defining the simulation types. These four classes correspond to four components in a simulation: Neurons, Synapses, Connections, and Layout. Neurons and Layout define the type of neurons and how neurons are arranged in a network, whereas Synapses and Connections control the connectivity of the network. Fundamentally, the Synapses component includes properties describing synapse behaviors, and Connections defines the rule of how synapses are formed and deleted.

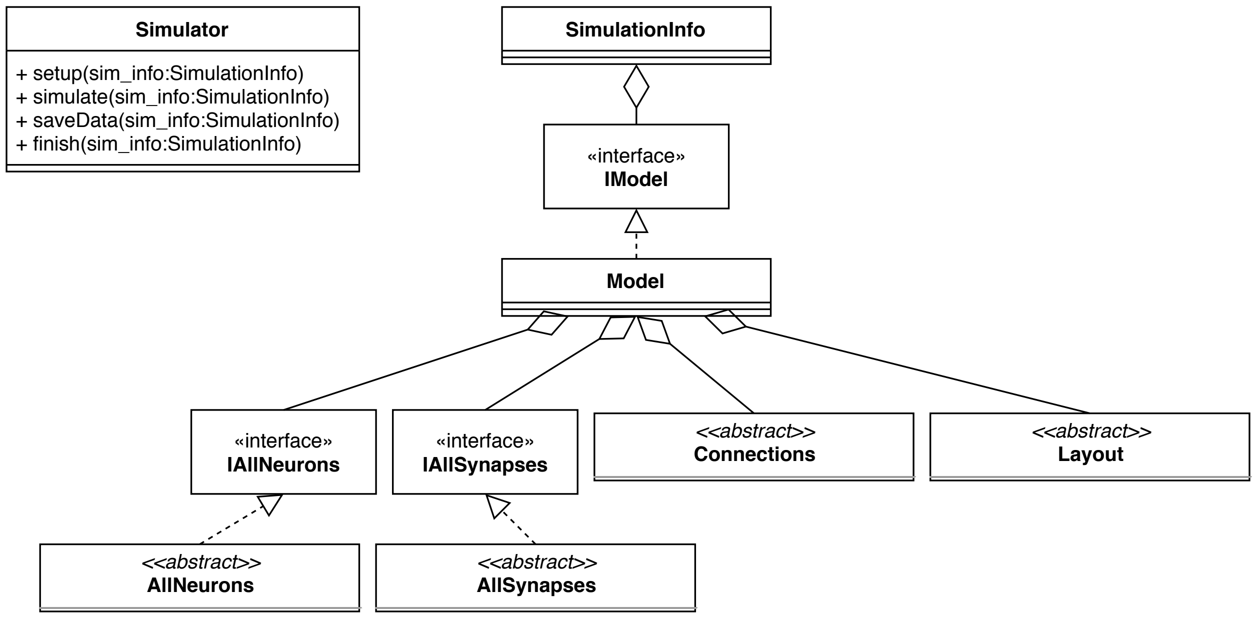


Figure 3: The key classes in BrainGrid. The SimulationInfo class stores network data; the Simulator class contains simulation operation methods. The SimulationInfo class has a Model class, which defines four components in a simulation: Neurons, Synapses, Connections, and Layout.

4.1.2. BrainGrid Simulation Process

The BrainGrid simulation process [14] (Figure 4) is described as follows:

1. User inserts at the command line to specify an input XML parameter file and the location of the result file (output file). The input file includes simulation settings and selected models (Neurons, Synapses, Connections, and Layout) (step 1 in Figure 4). The simulation settings include the number of neurons in a simulation, the length of simulation, etc. The selected models are neuron type, synapse type, connection type (growing connection or fixed connection), and the layout of neurons in a simulation.
2. The program starts with parsing the command line information and the input XML file. This information is saved in the simInfo object and is used to instantiate all network objects (Neurons, Synapses, Connections, and Layout) (step 2-6 in Figure 4).
3. After all objects are created, the simulator object begins to conduct the simulation operations (step 7-11 in Figure 4). This includes setting up objects by assigning values, performing simulation, saving result data, and deallocating objects to finish simulation.
4. Lastly, the result file will contain simulation data for the user.

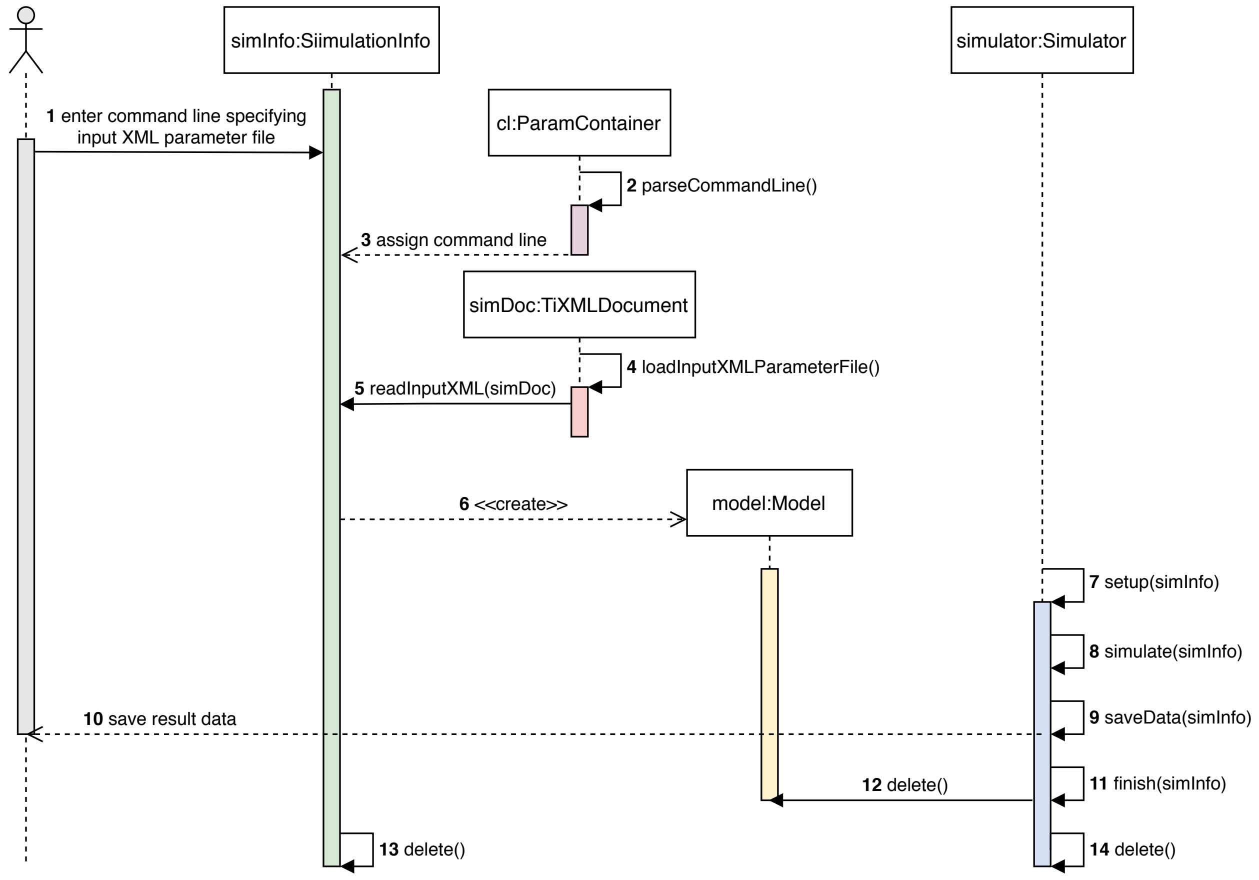


Figure 4: The BrainGrid simulation process.

As seen in Figure 4 step 8, the actual simulation operation was implemented in the *simulate()* method in the Simulator class. In the simulate() method, the simulation proceeds as a sequence of epochs. Within each epoch (Figure 5), neurons and synapses are updated every 0.1ms (one time step) while keeping the connectivity of the network constant. Although neuron and synapse states are updated, no synapse is formed or deleted. Only existing synapse states are updated. The formation and deletion of synapse happens at the end of each epoch, called the *growth update*. During the growth update, synapses are formed, deleted, and their weights are adjusted based on the spiking activities within the epoch. Once the growth update finishes, the simulation will move to the next epoch and end with the growth update until all epochs are finished.

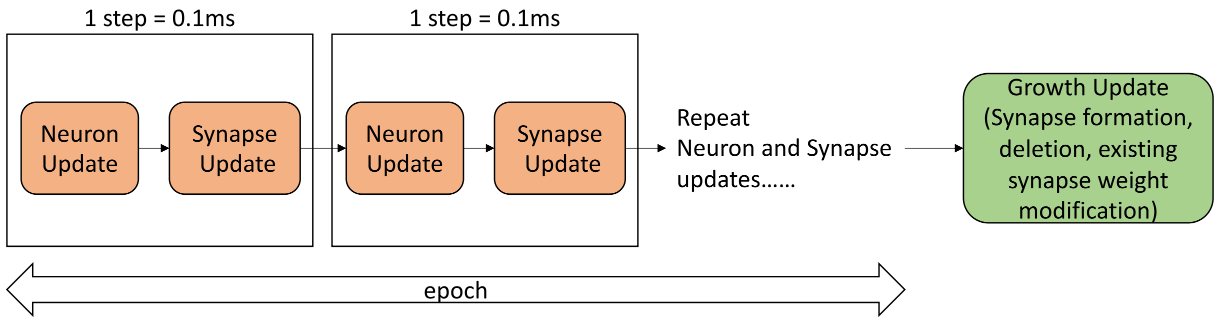

1 step = 0.1ms
1 step = 0.1ms
Neuron Update
Synapse Update
Neuron Update
Synapse Update
Repeat Neuron and Synapse updates……
Growth Update (Synapse formation, deletion, existing synapse weight modification)
epoch

Figure 5: The workflow in one epoch in a simulation. Within an epoch, the neuron and synapse are updated every 0.1ms (one time step). At the end of the epoch, the growth update is conducted.

4.2. The Cortical Culture Growth Model

The *cortical culture growth model* is a feature model [15] in BrainGrid which enables simulations of the growth of a network in dissociated cortical cell cultures. This model simulates 10,000 neurons growing 28 days *in vitro* (*DIV*). To demonstrate the nature of network dynamics and growth, this model includes dynamical neuron and synapse models and an activity-dependent neurite outgrowth model [15] to capture network characteristics.

4.2.1. Simulation Settings

The simulation layout (the Layout model) is designed to arrange neurons in a rectangular grid. A relatively small fraction of inhibitory and endogenous/spontaneous active neurons are distributed in this layout with the rest filled with excitatory neurons.

The simulation is conducted for a total of 600 epochs and each epoch is a 100-second activity. In each epoch, neurons and synapses are updated 1,000,000 times followed by a growth update. A total of 600 growth updates cause the network gradually evolves and result in a grown and stationary network.

4.2.2. Neuron and Synapse Model

Dynamical neuron and synapse models are adopted to simulate network dynamics. The neuron model is an *integrate-and-fire* type model [16]. In this model, the neuron membrane potential is determined by the synaptic input. If the potential is above the threshold, a spike is generated. After the spike, the neuron will undergo a refractory period before the next spike.

Similarly, the synapse model also demonstrates dynamics. This *dynamic* type model [17] involves activity-dependent facilitation and depression [15], [18]. This means neuron spiking activities have temporary impacts on the ability of synaptic transmission.

4.2.3. Neurite Outgrowth Model

The *neurite outgrowth model* [19], which is conducted during the growth update stage, defines how the network grows. In this model, each neuron has a region of connectivity. This is modeled as a circle with a radius that changes dynamically based on the neuron's spiking rate (Figure 6). This region simulates *neurite density*, which are axons and dendrites extended from the neuron's cell body. The model describes that low firing rates stimulate neurite outgrowth, and high firing rates cause regression [15]. Depending on the spiking rate, each neuron's region of connectivity may increase or decrease as the simulation continues.

When the regions of two neurons overlap, these two neurons are defined as connected. This causes the formation of the synapse. The overlapping region then becomes the synaptic weight, or synaptic strength. However, when two neurons do not overlap, no connection is defined between these neurons, and thus no synapse formed. This definition describes the network connection dynamics, illustrated in Figure 6.

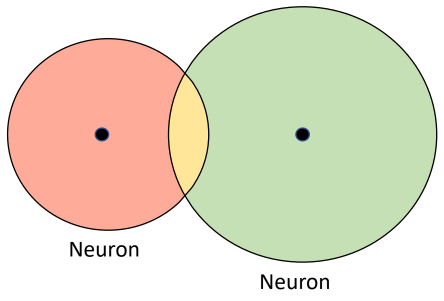


Figure 6: The definition of synaptic weight (strength) in neurite outgrowth model (re-drawn from [20]). The pink circle is the connectivity region of one neuron and the green circle is the region for another neuron. The overlapping yellow region is the synaptic weight. Depending on the spiking rates, the region of connectivity changes dynamically, causing synaptic weight to also change dynamically.

4.3. STDP Model in BrainGrid

The cell culture growth model successfully simulates the growth process of a network by using integrate-and-fire neurons, dynamic synapses, and the neurite outgrowth model. Using this model also leads to observations of some network behaviors like bursting. Nevertheless, this simulation only describes the facilitation and depletion dynamics of synaptic resources. It does not include another important development factor, i.e. spike timing, in synapse modification. To incorporate spike timing to the simulation, a spike-timing-based adjustment, or, STDP model, should be involved.

4.3.1. STDP Model Design

The STDP model in BrainGrid is implemented as a new synapse class. According to the STDP mathematical model [8], if there is a presynaptic spike and a postsynaptic spike and their time interval is within acceptable range, the synaptic weight is adjusted. This model definition is different from the neurite outgrowth model where the weight modification is conducted periodically at the end of each epoch. The weight adjustment should be conducted immediately when finding a valid spike pair during the epoch, not at the growth update stage. Therefore, the STDP model is defined as a type of synapse class in BrainGrid simulator instead of a new growth or connection class.

4.3.2. STDP Simulation Design

Figure 7 presents the design workflow of the STDP simulation in a development network. To simulate both the neurite outgrowth and STDP aspects, we separated these two development rules into two simulations and used serialization and deserialization, a programming technique for object storage, for continuing these simulations. In the first simulation, the cortical cell culture growth model was used to grow a network. At the end of the simulation where the network has connections established, serialization was performed to save the grown network to a file. Next, the second simulation began from deserializing the network. The STDP model was involved at this point for weight modification. Since the design involves serialization and deserialization, the implementation should have both serialization/deserialization and STDP class, which are explained in the next section.

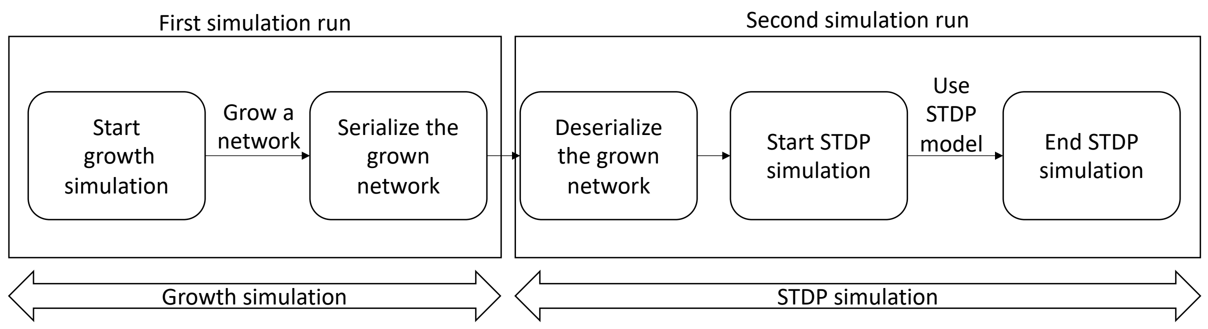


Figure 7: The design workflow of STDP simulation. At first, a growth simulation is conducted. Once the network is grown and serialized, the STDP simulation can begin from this point.

5. Method: Serialization and Deserialization of Large-Scale Neural Simulations

5.1. Serialization and Deserialization

Serialization and deserialization is a process for preserving objects in programming. It is used when an object needs to be stored or transmitted temporarily for later usage. In programming, objects are stored using data structures, but hardware and network infrastructure use bytes in processing data. When saving an object to files or sending an object over a network, there should be a mechanism for converting an object into a byte stream, and vice versa. This mechanism is called serialization and deserialization.

In some programming languages, serialization is provided in their standard libraries. For example, in Java, serialization can be achieved by implementing the *Serializable* or *Externalizable* interface [21]. For programming languages that do not have a built-in serialization feature, other various tools and techniques can be used. For instance, in C++, a manual approach would be using *memcpy()* to serialize objects. Libraries such as *Boost* [22] or *Cereal* [23] can also be used for object storage. In fact, there are also tools that support serialization in different languages. For example, *Protocol Buffers* [24] can be used in Java, Python, C++, etc. Basically, a user writes information about serializing objects in a .proto file, and depending on the language, a compiler will generate code which can be used in different language programs. In summary, since there are diverse serialization methods, performance, data size, and support are all varied.

5.2. The Cereal Library

To implement serialization and deserialization in BrainGrid, Cereal [23], an open-source C++ 11 serialization library, was utilized. There are several benefits of Cereal:

- It is built with C++ 11 standards, which is the same as the current BrainGrid simulator.
- It works on many C++ compilers, such as g++ and clang++.
- It is header only and has no external dependencies, so using this library only requires including files in the project.
- It can archive objects in various formats, including binary, XML and JSON.
- It provides support for most object types in C++ standard library.

- It is lightweight, fast, and easy to use.

Listing 1 and Listing 2 demonstrate an example of serializing and deserializing an object using the Cereal library. In this example, an object named samepleObject is serialized and deserialized in main.cpp (Listing 1). During serialization, an output archive is first created and the sampleObject is passed into the archive. Since sampleObject is MyClass data type, Cereal will look for the *serialize()* function in the MyClass.h file and check which data member needs to be serialized. As shown in Listing 2, the MyClass class has three data members. They are all passed into the *archive()* in MyClass::serialize() function, so Cereal will serialize these three members accordingly. Similarly, during deserialization, an input archive is created first. Next, the sampleObject is passed into the archive() method. Cereal will then read the data from the input stream and copy data back to the sampleObject. It is worth noting that the Cereal serialization function implementation in classes is within a .h file and not a .cpp file. Because serialization function is a template method, compiler requires the implementation details, so it can compile the function for a specific data type [25].

main.cpp

```
#include <cereal/archives/xml.hpp> //specify serial type
#include <fstream>
#include ”MyClass.h”
int main()
{
        //serialization
     {
        ofstream memory_out (serialization_file_name);
        cereal::XMLOutputArchive archive(memory_out); //create output archive

        MyClass sampleObject;

        archive(sampleObject); //write data to archive
     }

        //deserialization
     {
        ifstream memory_in (serialization_file_name);
        cereal::XMLInputArchive archive(memory_in); //create input archive

        MyClass sampleObject;

        archive(sampleObject); //read data to archive
     }
}
```

Listing 1: An example of Cereal serialization implementation. In the main.cpp file, a MyClass type object named sampleObject is serialized (top) and deserialized (bottom) using the Cereal archive(). Cereal will look for the serialization function in the MyClass class (see Listing 2) to serialize/deserialize its data.

MyClass.h

```
#include <cereal/types/vector.hpp> //cereal vector header
#include <vector>

class MyClass
{
     int a, b;
     vector<int> c;

     template <class Archive>
     void serialize( Archive & archive )
     {
        archive(a, b, c);
     }
};
```

Listing 2: An example of Cereal serialization implementation. The MyClass.h file serializes three data members by using the Cereal serialize() function.

Listing 1 and Listing 2 example is just one way of Cereal implementation. Depending on different cases, there are a variety of implementation options in Cereal. For example, the serialization function can be implemented not only as one serialize() function but also as split *save()* and *load()* functions. Another example is when object type involves inheritance or polymorphism, Cereal provides macros so these functionalities can be achieved.

Although Cereal supports serialization in many aspects, its main drawback is that it can only serialize smart pointer data types, not primitive raw pointers. If a data member is a raw pointer, doesn't matter it points to an object or dynamically allocated array, Cereal can not serialize any of them. An alternative solution should be utilized to cope with this issue, which is explained in the later section.

5.3. Serialization and Deserialization in BrainGrid

5.3.1. Implementation Challenges

Because the purpose of serialization is providing a grown network for the STDP simulation, the serialized data should represent the whole network. At first, we attempted to serialize all network components. Starting from the main driver in BrainGrid, we examined each object to see whether or not it was a network component. Cereal serialization function was added to its class if so. Next, in this class, all data members were examined for serialization. If any data member needed to be serialized, a serialization function was also added to the data member's class type. We continued to investigate all objects until all network objects were captured. Although this approach may work, it was difficult to completely implement this approach. One reason is because BrainGrid included more than 50 classes and had more than 80 objects created in one simulation. Figure 8 demonstrates a simple object diagram in BrainGrid. In each object, there were various attributes containing different pieces of information. To understand the functionality of each attribute and determine if it contained information that should be preserved were difficult in such a complex program. In addition, raw pointer serialization was another problem. In BrainGrid, a lot of pointers and references were used. Serializing them were not supported by Cereal. The alternative solution would be either modifying existing code to change to the smart pointer or to use dereference to pass in the object itself. Changing to a smart pointer involved a lot of code changes, and dereferencing pointers did not work in polymorphic objects.

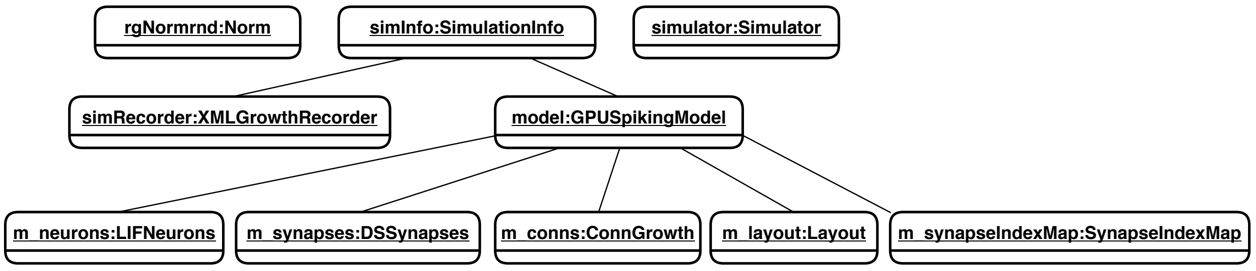


Figure 8: A basic BrainGrid object diagram. Each object in this diagram had various attributes with different pieces of information to be examined for serialization.

Considering the complexity of the program and the limitation of Cereal, we used a different approach to implement serialization. Instead of serializing all network components, only four essential objects were selected: synapse weights, synapse source neurons, synapse destination neurons, and neuron radii.

The first three objects were selected because they represented the connections of a network. If viewing the network as a weighted graph, synapses would be edges and neurons would be vertices. These are definitely essential network elements. In addition, neuron connectivity radii was also selected. This is because in the BrainGrid growth model, connectivity radii were used to determine the synaptic weight. If using the growth model and only serializing weights, during deserialization, even weights were deserialized successfully, these deserialized weight data would still be replaced by the radii computed weights. Thus, connectivity radii should also be serialized if using the growth model.

5.3.2. Cereal Serialization and Deserialization

To implement serialization on these four selected objects, the Cereal save() and load() functions, were utilized. Figure 9 demonstrates these four objects and the classes they reside in. Synaptic weights, source neurons, and destination neurons are attributes in the AllSynapses class; neuron radii is an attribute in the ConnGrowth class.

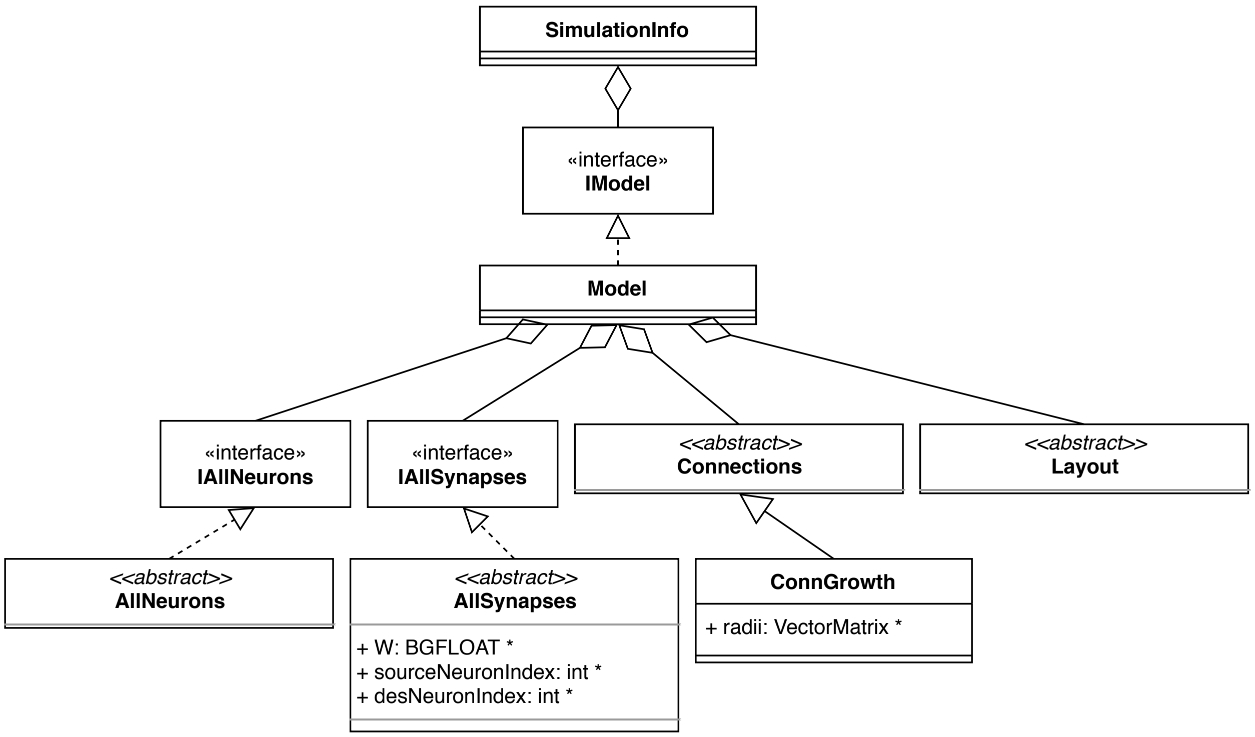


Figure 9: The class diagram demonstrates the classes these four objects in. Synaptic weight, source neuron, and destination neuron are in the AllSynapses class. Neuron radii is in the ConnGrowth class.

The synaptic weight, source neuron, and destination neuron are designed as synapse properties in the AllSynapses class. As illustrated in Figure 10, all synapse models inherit synapse properties from this class. Basically, these synapse properties are all stored as dynamic allocated arrays. Figure 11 illustrates synaptic weight and some other synapse properties in dynamic arrays. Each element in these arrays represents one current state of a synapse. For a specific synapse, all of its states are stored in the same index position in each array. In other words, these dynamic arrays are all with the same size. The size of the array is determined by *maxSynapsePerNeuron*, a variable with a number value from user input, multiplied by total number of neurons in a simulation. For each neuron, there are a number of spots reserved for the neuron to store its synapses. That number is the value in maxSynapsePerNeuron variable.

Each array stores synapses in the order of destination neurons. In other words, from index position 0, the first group of spots is reserved for destination neuron 1, the next group of spots is reserved for destination neuron 2, and so on. This rule applies to all arrays, so one synapse's states can be found in the same index position.

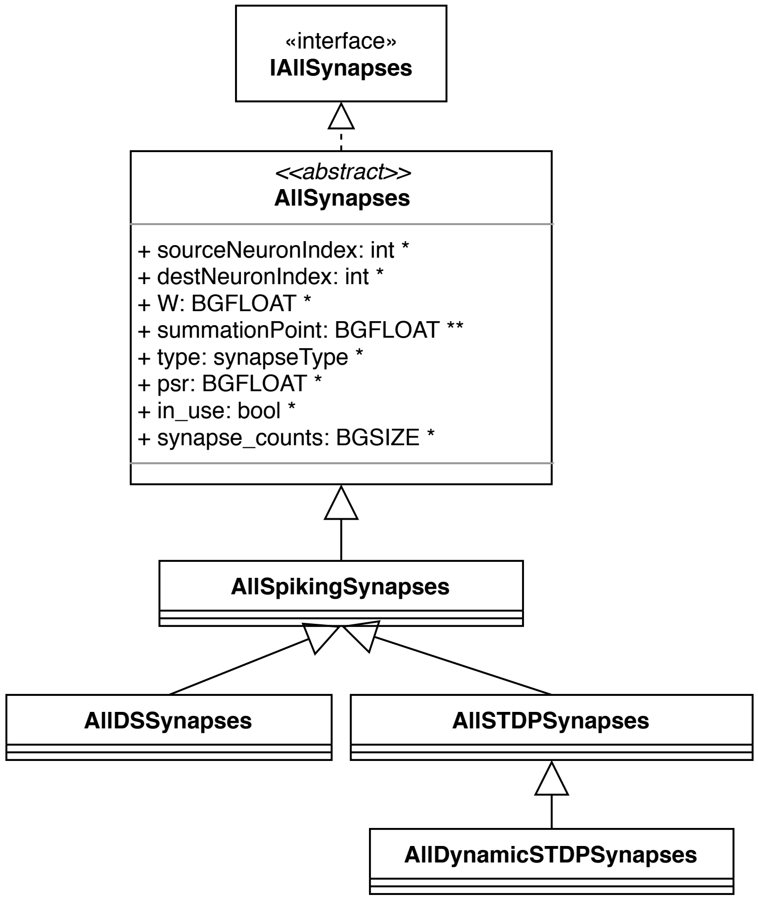


Figure 10: Synapse property attributes in the AllSynapses class. Synaptic weight (W), source neuron (sourceNeuronIndex), and destination neuron (destNeuronIndex) are all property attributes in this class. All synapse models inherit properties from this class.

Since synapses in a network are not a constant component and these synapse property arrays only store current existing synaptic states, some elements in arrays may not be used at a given time. An array named "in_use" is utilized to indicate whether an index position is in use and has information from an existing synapse. For example, if an index position shows False in the "in_use" array, this position is not currently used. This means the value stored in this position can be ignored. This also indicates that this position in the synaptic weight array should have a zero value. A zero value in the weight array also means its position is not in use.

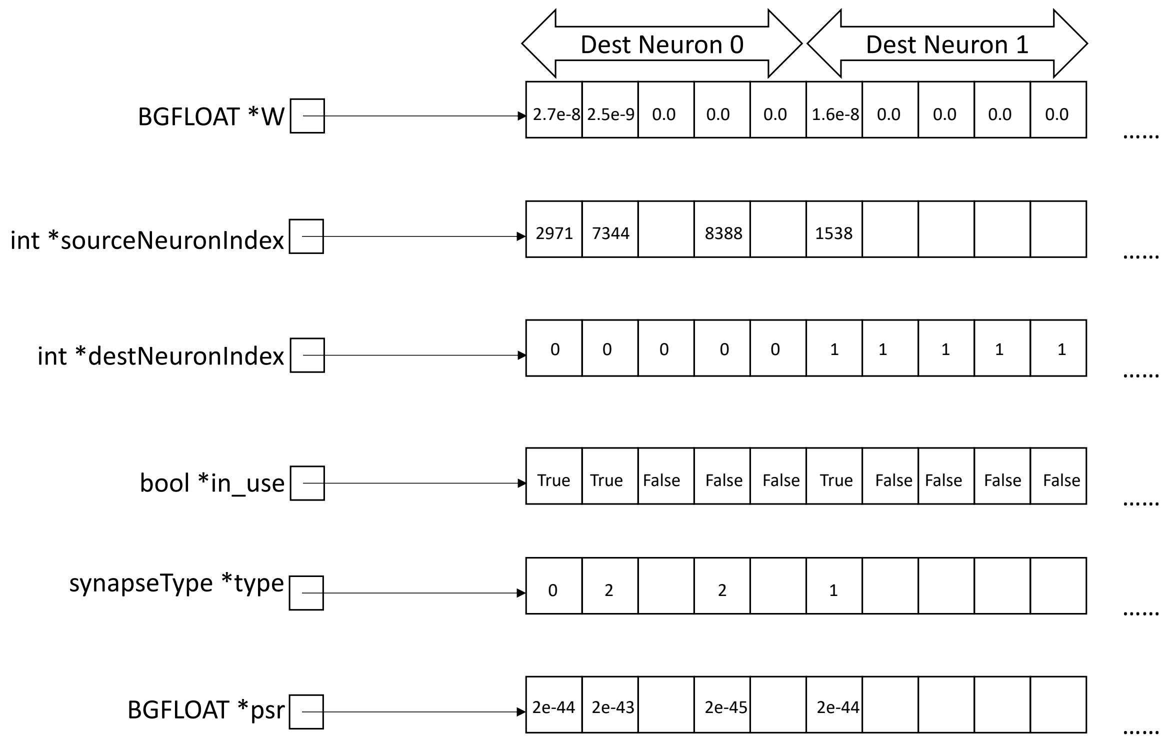


Figure 11: Synapse properties are stored in dynamic allocated arrays. Each property is one array (from top to bottom, synaptic weight (W), source neuron (sourceNeuronIndex), destination neuron (destNeuronIndex), in_use array (in_use), synaptic type (type), and postsynaptic response (psr)). The current states of one existing synapse are located in the same index position in each array.

Since BrainGrid stores the states of synapses in different objects, it was difficult to determine what object should be serialized at first. Although serializing all state objects could be an option, states like postsynaptic response were tough to handle at deserialization for continuing the simulation. As a result, we chose synaptic weight and source and destination neurons. These were the most representable elements of the network for serialization.

Nevertheless, serializing dynamic arrays was another issue. As mentioned above, since the Cereal library does not support dynamic array serialization, an alternative method should be utilized. In this project, we used a helper data structure to solve this issue. Figure 12 describes the algorithm to serialize and deserialize dynamic arrays using a helper data structure *Vector*. In save() method, vectors were created first, and values in dynamic arrays were copied over. As a result, the objects were serialized as a vector data type. On the other hand, when deserializing objects in load() method, vectors were created first. The data in the serialization file was copied over. Finally, values in vectors were copied back to dynamic arrays.

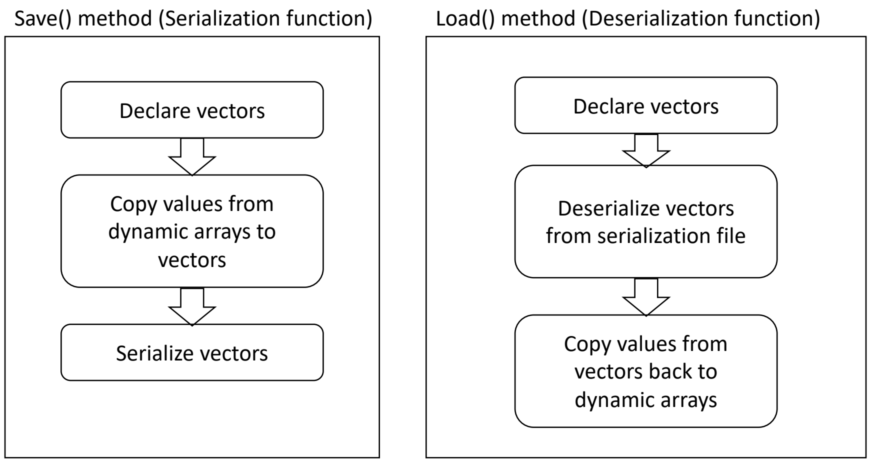

Save() method (Serialization function)
Declare vectors
Copy values from dynamic arrays to vectors
Serialize vectors
Load() method (Deserialization function)
Declare vectors
Deserialize vectors from serialization file
Copy values from vectors back to dynamic arrays

Figure 12: The serialization and deserialization algorithm when dealing with dynamic array data types. Vector is a helper data structure.

Similarly, the neuron connectivity radii object is also stored in a dynamic array. In the ConnGrowth class, radii is a raw pointer of VectorMatrix data type, which is a tool class storing 1D dynamic array data. Since each neuron has one radius, the size of the array is the total number of neurons. Serialization implementation for radii also used a helper vector, similar to the synaptic weight object, which is illustrated in Figure 12.

5.3.3. Serialization and Deserialization WorkFlow in BrainGrid

Figure 13 presents the workflow of adding serialization and deserialization features in BrainGrid. Serialization is conducted after the simulation is completed and before objects are deallocated. On the other hand, deserialization is conducted after objects are instantiated and before the simulation starts.

At the beginning of the simulation, the user can choose whether to perform serialization, or deserialization, or both at the command line. Both serialization and deserialization are optional to users when conducting a simulation.

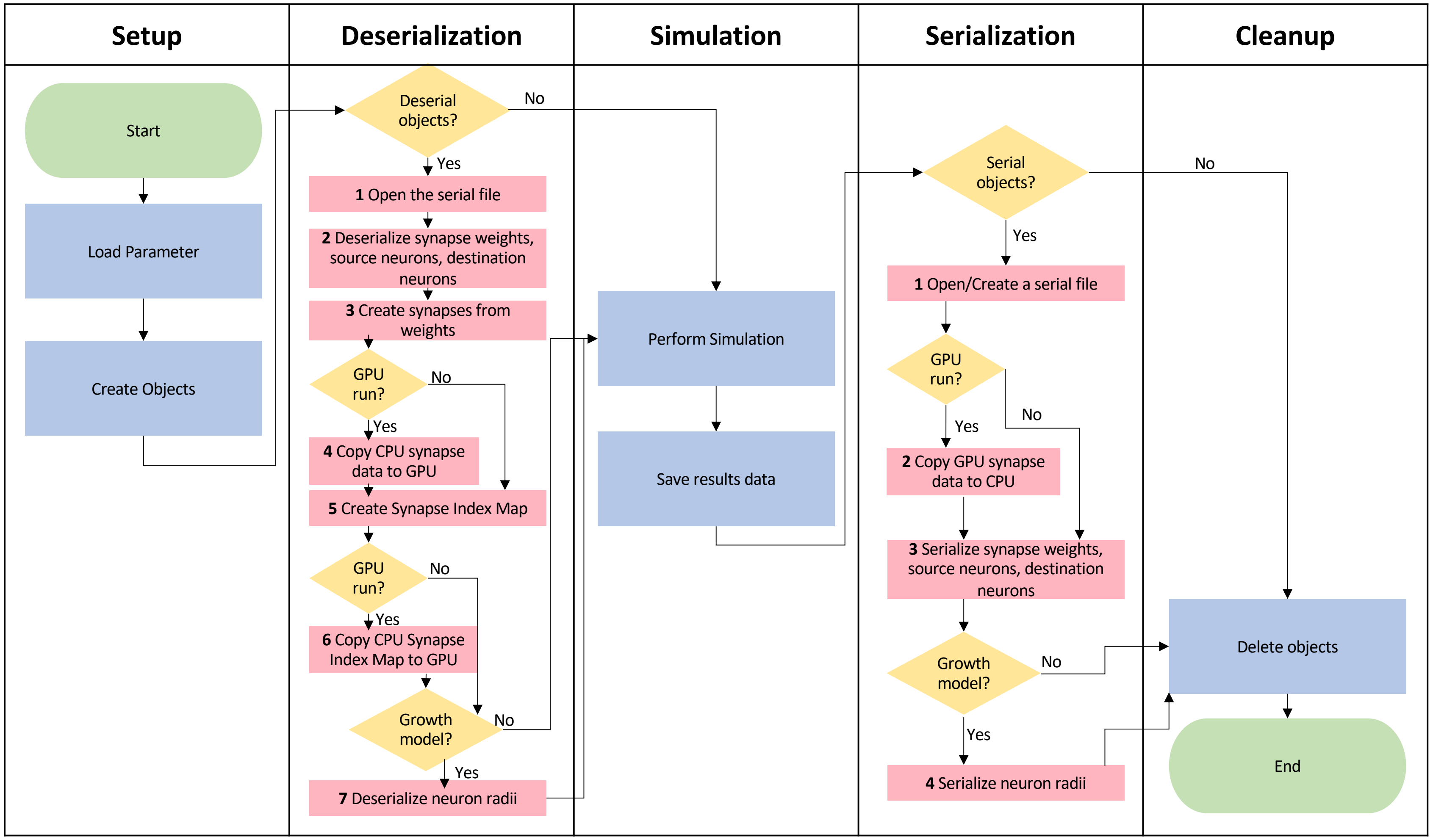

Setup
Deserialization
Simulation
Serialization
Cleanup
Start
Load Parameter
Create Objects
Deserial objects?
No
Yes
1 Open the serial file
2 Deserialize synapse weights, source neurons, destination neurons
3 Create synapses from weights
GPU run?
No
Yes
4 Copy CPU synapse data to GPU
5 Create Synapse Index Map
GPU run?
No
Yes
6 Copy CPU Synapse Index Map to GPU
Growth model?
No
Yes
7 Deserialize neuron radii
Perform Simulation
Save results data
Serial objects?
No
Yes
1 Open/Create a serial file
GPU run?
No
Yes
2 Copy GPU synapse data to CPU
3 Serialize synapse weights, source neurons, destination neurons
Growth model?
No
Yes
4 Serialize neuron radii
Delete objects
End

Figure 13: The workflow of serialization and deserialization in BrainGrid.

Figure 14 demonstrates all functions implemented or utilized in this workflow (Figure 13). As seen in Figure 14, implementing this workflow involves not only Cereal serialization functions but also other helper functions. For example, in step 2 (Figure 13) during serialization and step 4 (Figure 13) during deserialization, a *copyGPUSynapseToCPU()* function and a *copyCPUSynapseToGPU()* function were implemented and called for GPU-based simulation. If choosing a GPU-based simulation, some computations are conducted on the GPU and the data is also on the GPU. Thus, to serialize and deserialize the GPU data, the data needs to be copied to the CPU for serialization and to the GPU for deserialization. These two functions were implemented in the GPUSpikingModel class. Since the GPUSpikingModel class contains a pointer data member which points to the corresponding GPU location storing synapse attribute data, implementing the function in this class can retrieve data in GPU.

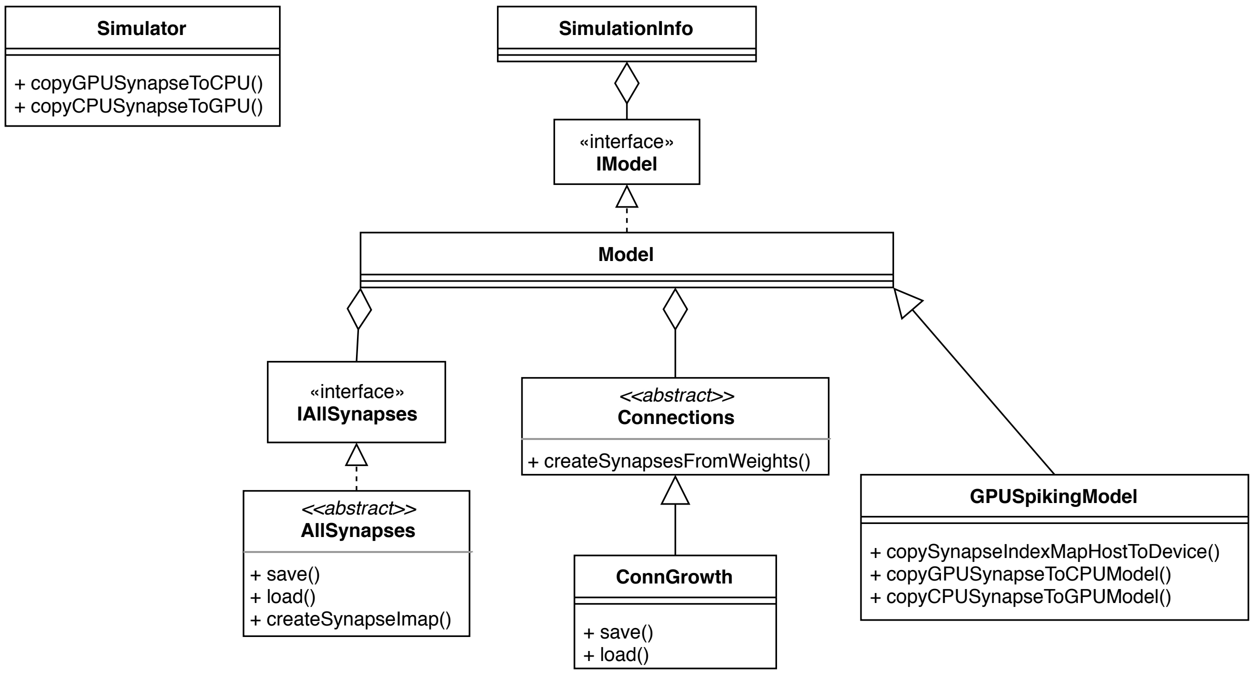


Figure 14: All functions implemented or utilized in the serialization and deserialization workflow.

In step 3 (Figure 13) during deserialization, a *createSynapsesFromWeights()* function was implemented in the Connections class and called to re-create synapses. This was because the three serialized synapse objects only represent some of the synapse properties. Other synapse properties such as synaptic types need to be re-constructed using deserialized synaptic weight data. Listing 3 shows the createSynapsesFromWeights() function. The function iterates each element in the deserialized synaptic weight array. If an element has a value which is not zero, it means there is an existing synapse. Then, this synapse will be re-created by calling a *createSynapse()* function. Lastly, in step 5 and step 6 during deserialization (Figure 13), the function to create synapse index map and copy it to the GPU was called for reconstructing the simulation infrastructure, so the simulation can use the newly re-created synapses to continue the operation.

```
for (index = 0; index < synapse_weight_array.size(); index++) {
        if (synapse_weight_array[index] is not zero) {
                // get values from serialized data
                float weight = synapse_weight_array[index];
                int source_neuron = synapse_source_neuron_array[index];
                int destination_neuron = synapse_destination_neuron_array[index];

                //create the synapse by calling createSynapse() function to assign default synapse values
                createSynapse(weight, source_neuron, destination_neuron)
        }
}
```

Listing 3: The function to re-create synapses from weights. After the synaptic weight, source neuron, and destination neuron are deserialized, the createSynapsesFromWeights() function is called to iterate through the synaptic weight array to see if an element with a value is not zero. If True, it means an existing synapse and this synapse will be re-created by calling the createSynapse() function, an existing function to assign all synapse property attributes with a default value.

6. Method: STDP Model

6.1. STDP Synapse Class and Simulation Models

The STDP synapse class used in this project is based on an existing class implemented previously. We followed the STDP mathematical model presented in [8] and modified this existing class to simulate the basic STDP behavior.

As mentioned above, since the design of the STDP simulation workflow involved two simulations and each simulation represented different phases in network development, different models were selected in each simulation run. Figure 15 presents models used in two simulations. In the growth simulation, the growth model (Connection) and the dynamic synapse model (Synapse) were used. In the STDP simulation, the static model (Connection) and the STDP synapse model (Synapse) were used. The static model was different from the growth model where synapses and their weights in this model were defined as input. They remained the same throughout the entire simulation. Therefore, by using the static model and the deserializing synapses from the growth model, the synaptic weight modification will be only based on STDP learning rule, so the impacts of STDP can be observed.

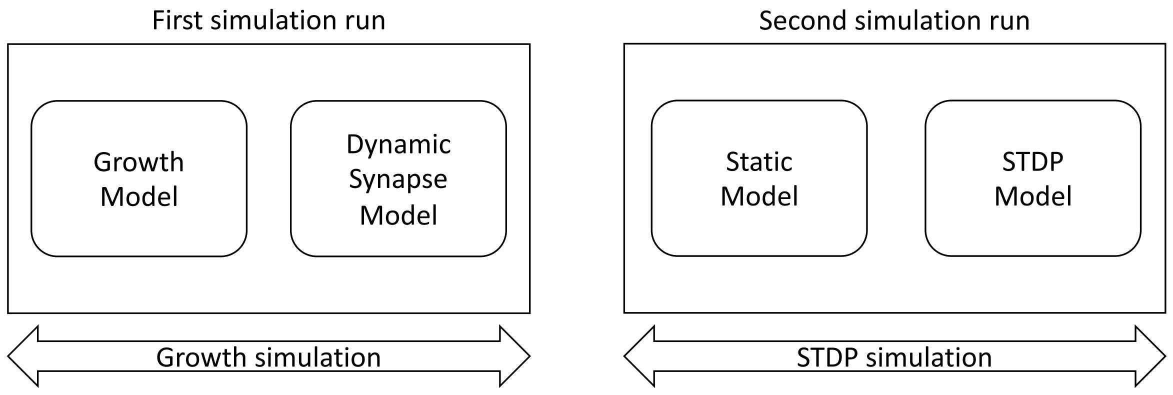


Figure 15: The models used in STDP simulation.

6.2. STDP Implementation

The existing STDP synapse class was implemented as the derived class of the *AllSpikingSynapses* class (Figure 10). In this class, two member functions were used to conduct synaptic weight adjustment: the *advanceSynapse()* method and the *stdpLearning()* method.

The advanceSynapse() method was an overwritten method which had the signal transmission logic for each synapse performed in a single time step during epoch (the synapse update step in Figure 5). Fundamentally, each synapse class overwrote this method to have its own logic to receive signals from a presynaptic neuron, calculate the postsynaptic response, and send the calculated signals to a postsynaptic neuron. Differing from other synapse classes, the advanceSynapse() method in the STDP synapse class involves not only signal transmission logic but also the weight adjustment process.

Figure 16 presents the advanceSynapse() method algorithm in the STDP synapse class. The method starts by checking if the synapse is an inhibitory synapse (its presynaptic neuron is an inhibitory neuron) or a zero weight synapse. This is because according to [8], the STDP learning only applies to excitatory neurons. If it is an inhibitory synapse or a zero weight synapse, it will only conduct the signal transmission. There is no weight adjustment involved.

Next, another check is conducted to see whether or not the synapse has a presynaptic spike that needs processing. This means that a presynaptic spike is generated previously, and now the signal is delivered to the synapse to be transmitted to the postsynaptic neuron. If this is the case, then the method will retrieve the latest preceding spike time from the postsynaptic neuron (the last time the postsynaptic neuron generated a spike), and calculate the time interval between the current time (the spike from the presynaptic neuron) and the latest preceding spike (a spike from the postsynaptic neuron). If the time interval is within the STDP learning window, this interval is then used to adjust weight by entering the stdpLearning() method. After one adjustment finishes, the algorithm moves to the preceding spike from postsynaptic neuron before the latest one, calculating another time interval (using current time as the presynaptic spike and preceding spike time as the postsynaptic spike). If this interval is still within an acceptable range (in the learning window), the stdpLeanrning() method is called again to perform another weight adjustment routine. The algorithm will continue to retrieve preceding spikes to see if the intervals are acceptable for entering the learning method until the interval is out of the learning window.

Similarly, the algorithm also checks whether or not the synapse has a postsynaptic spike that needs processing. This means that a postsynaptic spike is generated previously and now the signal is backpropagated (delivered in reverse) to the synapse to be transmitted to the presynaptic neuron. If this is the case, the method will do the same thing: retrieving the preceding spike times from the presynaptic neuron, calculating the time intervals, and entering stdpLearning() accordingly.

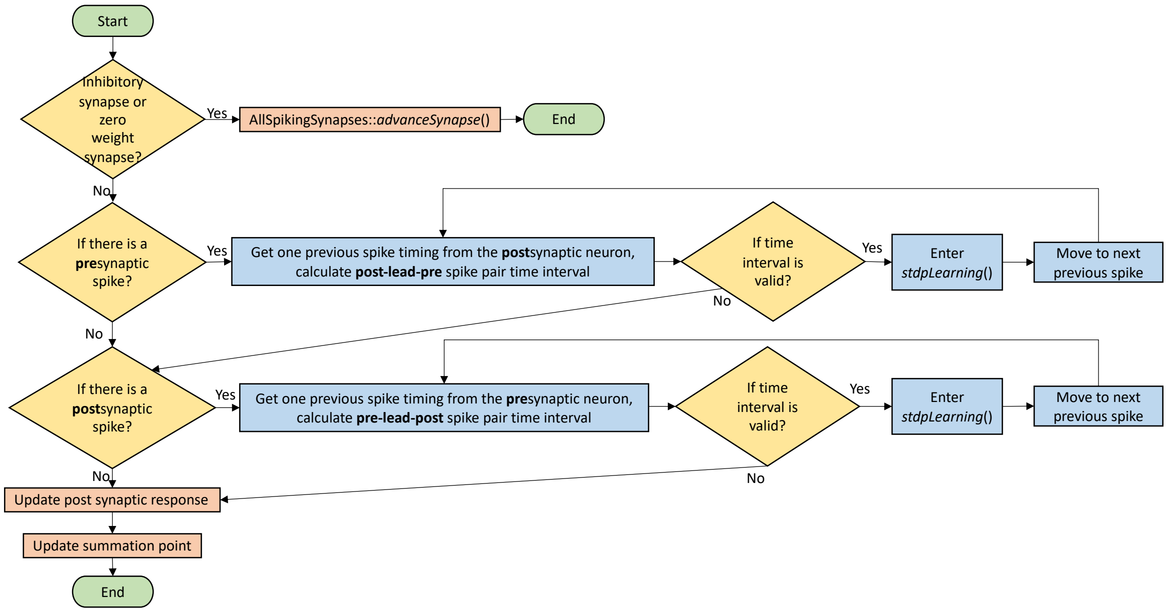

Start
Inhibitory synapse or zero weight synapse?
Yes
AllSpikingSynapses::advanceSynapse()
End
No
If there is a presynaptic spike?
Yes
Get one previous spike timing from the postsynaptic neuron, calculate post-lead-pre spike pair time interval
If time interval is valid?
Yes
Enter stdpLearning()
Move to next previous spike
No
No
If there is a postsynaptic spike?
Yes
Get one previous spike timing from the presynaptic neuron, calculate pre-lead-post spike pair time interval
If time interval is valid?
Yes
Enter stdpLearning()
Move to next previous spike
No
No
Update post synaptic response
Update summation point
End

Figure 16: The algorithm in AllSTDPSynapses::advanceSynapse() method.

Figure 17 illustrates an example of the process of retrieving preceding spike times and calculating time intervals to enter stdpLearning(). In this example, the synapse has a presynaptic spike that needs processing. The algorithm starts by retrieving the latest preceding spike from the postsynaptic neuron and calculates the time interval (spike pair #1 in Figure 17). Since the interval is within the learning window, this interval enters the stdpLearning() method. The algorithm continues to find preceding spikes from the postsynaptic neuron to see if a time interval is within the range (spike pair #2 and #3 in Figure 17). The iteration stops at an interval which is out of the range (spike pair #4 in Figure 17).

The synaptic weights are adjusted in the stdpLearning() method (Listing 4). When a spike pair enters the stdpLearning() method, the algorithm starts by using the time interval to calculate the fractional change in weight. This fraction was added to 1.0 to become the scaling ratio. If the scaling ratio is less than zero, the ratio is reset to zero so the weight is adjusted to zero and remains at zero for the rest of the simulation. On the other hand, if the scaling ratio is larger than zero, the weight is adjusted by multiplying the scaling ratio. The algorithm ends with a final check to see if the weight is bigger than the maximum allowable weight. If that is the case, the weight will be reset to the maximum allowable weight.

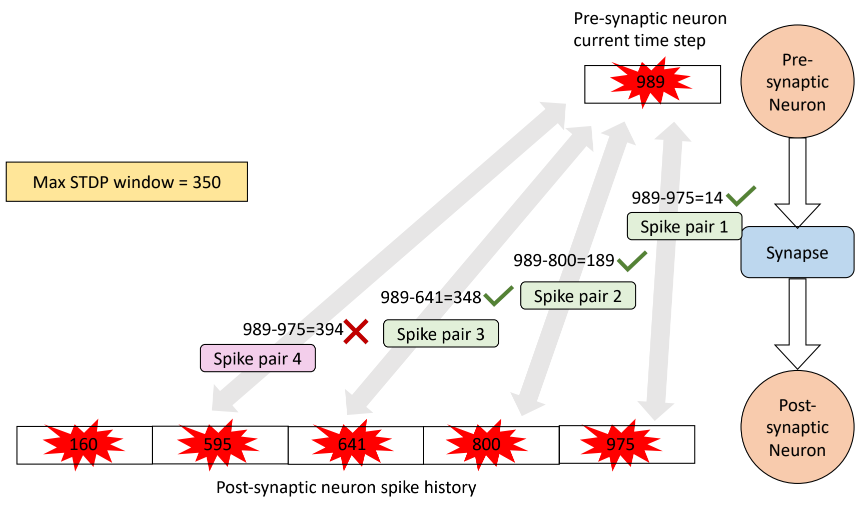


Figure 17: The example of the process of retrieving preceding spike times and calculating time intervals. Each number in a red explosion shape is the time step when a spike is generated.

```
if (presynaptic spike precedes postsynaptic spike(pre-leads-post spike pair)) {
        //Potentiation
        fractional_change = Apos * exp(-abs(time_interval)/taupos); //fractional_change is fractional
change in weight

} else if (postsynaptic spike precedes presynaptic spike(post-leads-pre spike pair)) {
        //Depression
        fractional_change = Aneg * exp(-abs(time_interval)/tauneg);
}

scaling_ratio = 1.0 + fractional_change; //add 1.0 to become weight scaling_ratio

//If scaling_ratio is negative, means weight is adjusted to zero and will never be changed.
//This means the synapse is deleted. No connection (synapse) anymore.
if(scaling_ratio < 0) {
        scaling_ratio = 0;
}

//Adjust weight
synaptic_weight = synaptic_weight * scaling_ratio;

//If synaptic_weight is bigger than max_weight, set it to max_weight
if(fabs(synaptic_weight) > max_weight) {
         synaptic_weight = max_weight;
}
```

Listing 4: The pseudocode of the stdpLearning() method.

7. Results

7.1. Serialization and Deserialization Verification

To verify serialization and deserialization features, eight experiments were performed (Table 1) on the BrainGrid multi-threaded version program to test different simulation scenarios. Scenarios varied in terms of simulations length, machine type, total neurons, and serialization file type.

Table 1: Serialization and Deserialization Verification Experiments

Experiment	Number of Epoch (Serialization Run)	Number of Epoch (Deserialization Run)	Number of Epoch (Whole Run)	Machine Type	Number of Neurons	Serialization File
1	10	10	20	CPU	100	XML
2	10	10	20	GPU	100	XML
3	10	100	110	CPU	100	XML
4	10	100	110	GPU	100	XML
5	10	100	110	CPU	100	Binary
6	10	100	110	GPU	100	Binary
7	100	100	200	GPU	10,000	XML
8	300	300	600	GPU	10,000	XML

Each experiment involved three simulations:

1. A growth simulation ended with serialization (serialization run)
2. A growth simulation began with deserialization from #1 (deserialization run)
3. A growth simulation which ran as long as the combination of #1 and #2, without serialization and deserialization (whole run)

All three simulations were conducted using the same input parameter file and the same machine. In each experiment, multiple files were generated for verification tests (Figure 18). Three verification tests were performed:

1. Objects are serialized correctly – comparing the serialization file with object state output after serialization. The output printed out object states to a text file. A Python script was written to retrieve data from a serialization file and the state output file to conduct a comparison. These two data items should be the same to pass the test.

2. Objects are deserialized correctly – comparing object state output before and after deserialization. Another state output file was generated after the deserialization run with the information of object states from before and after deserialization. A Python script was also used to retrieve data. To pass the test, the object states before deserialization should be different compared to after deserialization. In addition, the object states after deserialization should be the same as the serialization file.
3. The deserialization run generates similar results as a whole run – comparing whole run results with deserialization results. Since only four data members were serialized (rather than the entire state of the simulation), results of the deserialization run were not exactly the same as the whole run. However, it should be similar enough to prove that the deserialization works properly. Results were evaluated by visualizing two simulation results with their neuron radii and spiking rates within each epoch. The average and median radii and spiking rates in each epoch were also compared. Lastly, in each epoch, we selected a neuron that behaved the most differently in radius and spiking rate between the whole run and the deserialization run. We then plotted the differences to see if the maximum difference in radii and rates between the two runs changed during the entire simulation. The expected results should show similar maximum differences in each epoch for both radii and spiking rates. Figure 19 demonstrates an example of all plots generated in one experiment for this verification test.

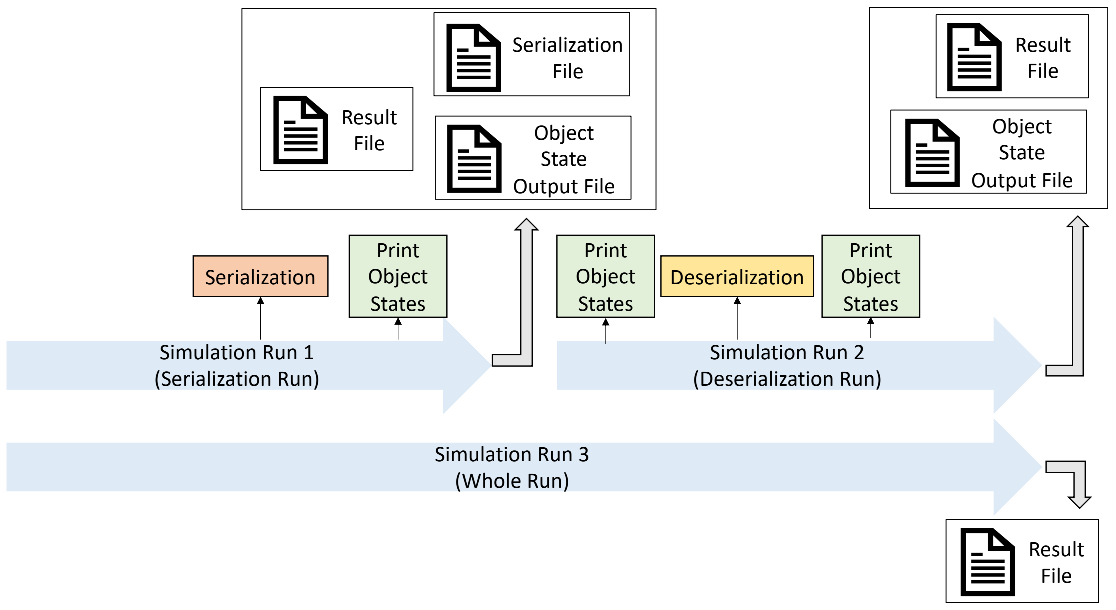


Figure 18: In each experiment, multiple files were generated for verification. Verification involves three parts: 1) Comparing serialization file and object states after serialization to ensure that serialization works properly. 2) Comparing object states from before and after deserialization to ensure deserialization that works properly. 3) Comparing deserialization run results and whole run results to ensure that deserialization generates similar simulation results as non-serialization results (whole run).

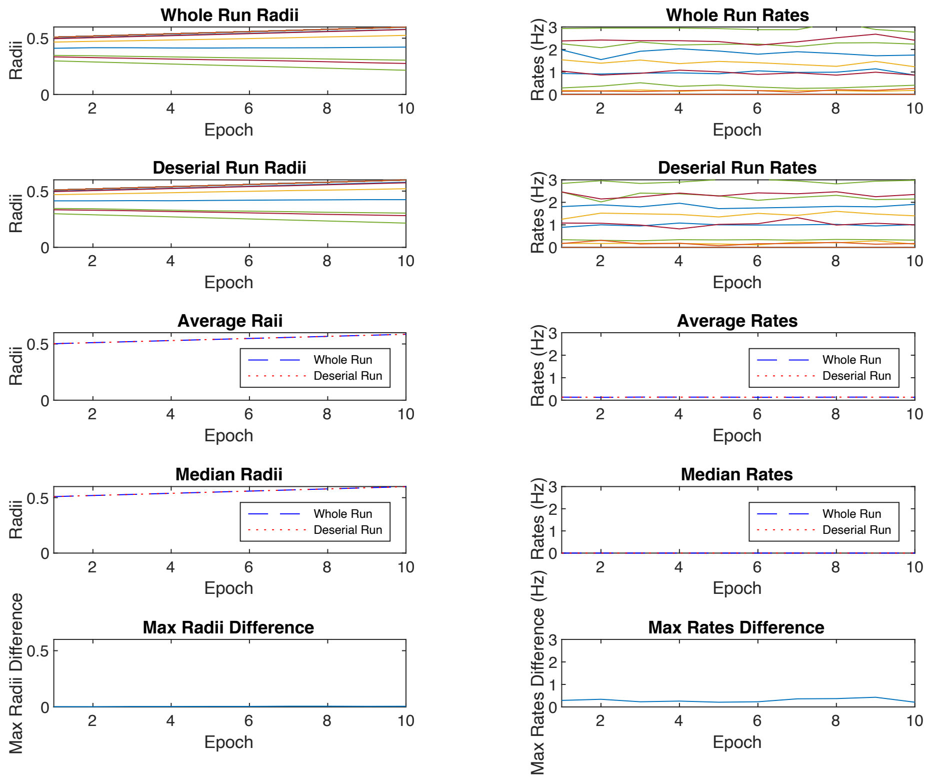


Figure 19: All plots generated in one experiment for verification test #3. Two simulation results were compared by visualizing neuron radii and spiking rates in each epoch.

7.1.1. BrainGrid Multi-threaded Version Verification

Table 2 shows the verification testing results. As seen in Table 2, all experiments were passed on the first two verification tests: correct serialization and deserialization. This indicated the data was successfully serialized and deserialized. However, in experiment 8, when comparing simulation results between the deserialization run and the whole run, the maximum differences of radii and spiking rates increased significantly around 120 epochs (Figure 20 C, D). In experiment 7 (Figure 20 A, B), when the simulation was 100 epochs long, the maximum differences of radii and rates remained similar for the entire simulation. However, in experiment 8, when the simulation became 300 epochs long, a significant increase was observed. This means that at least one neuron's spiking rate and at least one neuron's radius exhibited abnormal behavior at around 120 epochs. To further investigate this abnormality and see how many neurons present this behavior, we retrieved the data from the very last epoch and found out that among 10,000 neurons, there were 12 neurons that showed significant differences in radii. The same 12 neurons demonstrated significant differences in spiking rates as well. This finding indicated that the abnormal behavior came from the spiking rates because in the growth model, spiking rates were used to determine radii. Even though we knew the issue was from spiking rates, the reason for this behavior was still unknown. Thus, we performed experiment 8 one more time on the single-threaded version code to find out if this behavior was also observed and if so, what the possible cause of this issue might be. The results of testing on the single-threated code is demonstrated in the next section.

Table 2: Multi-threaded version verification testing results

Experiment	Verification Test #1 Correct Serialization	Verification Test #2 Correct Deserialization	Verification Test #3 Similar Results
1	Pass	Pass	Pass
2	Pass	Pass	Pass
3	Pass	Pass	Pass
4	Pass	Pass	Pass
5	Pass	Pass	Pass
6	Pass	Pass	Pass
7	Pass	Pass	Pass
8	Pass	Pass	Fail

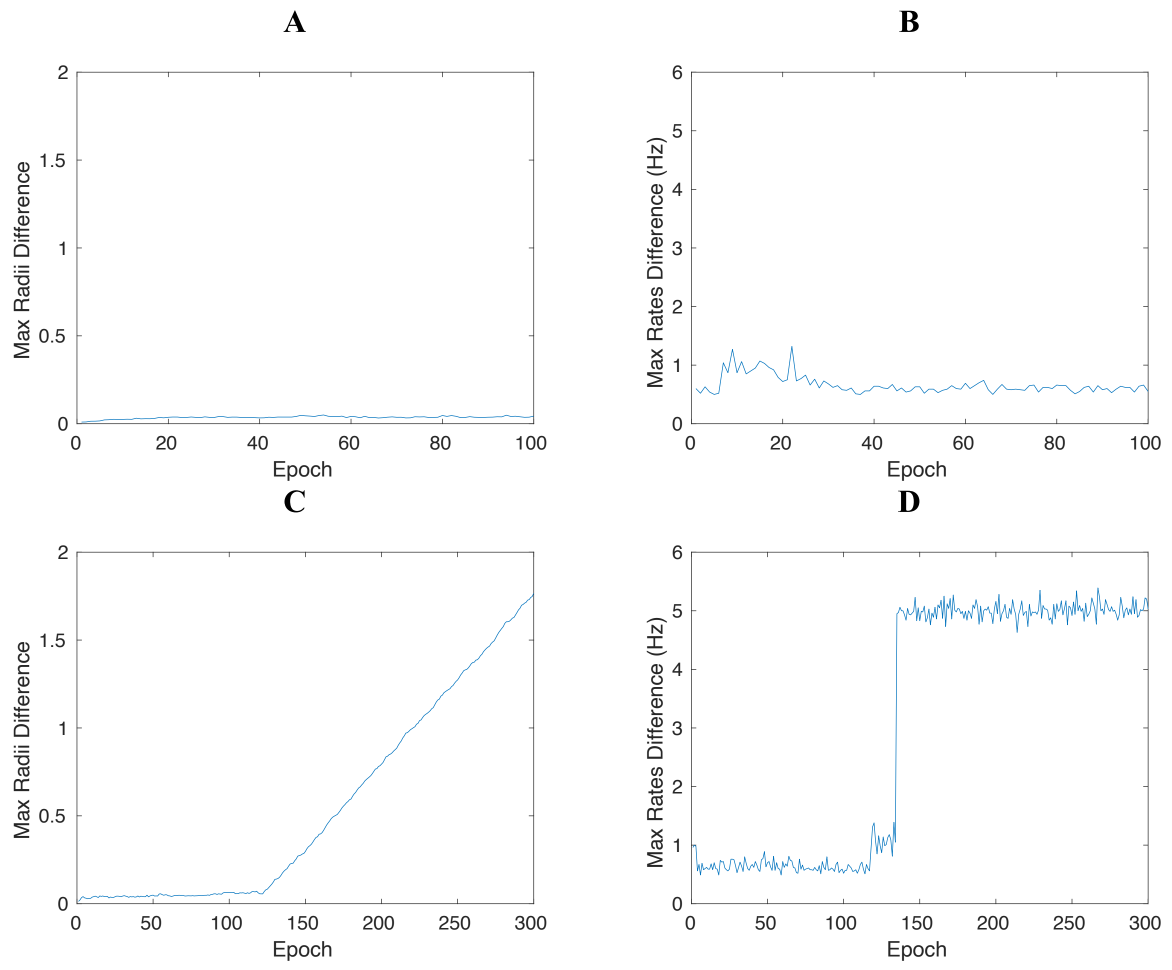


Figure 20: Maximum differences of radii and spiking rates in experiment 7 and experiment 8 on multi-threaded version of code. In experiment 7, maximum radii (A) and spiking rates (B) differences remained similar throughout the entire 100-epoch simulation; however, in experiment 8, maximum radii (C) and spiking rates (D) differences showed significant increases at around 120 epochs.

7.1.2. BrainGrid Single-threaded Version Verification

To further investigate the potential cause of abnormal spiking rates, one more experiment was performed on the single-thread version code. Figure 21 showed the results using the single-threaded version code. As seen in Figure 21, abnormal spiking rates were still observed, but the maximum differences were less than the multi-threaded version. The other difference was the timings. In the single-threaded version, an abnormality appeared at around 150 epochs, which was different from the multi-threaded version. In addition, only two neurons out of 10,000 neurons exhibited this abnormal spiking behavior. These two neurons' radii and rates in each epoch is shown in Figure 22. This figure showed that although two neurons behaved abnormally at different times, their behaviors were very similar—both neurons stopped spiking in the middle of the simulation, which caused their radii to begin to increase.

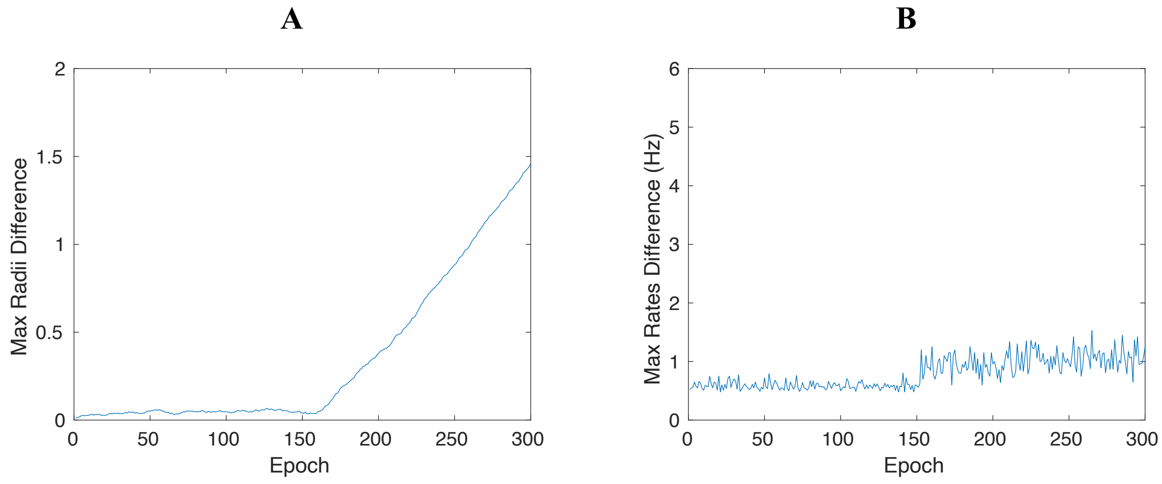


Figure 21: Maximum differences of radii (A) and rates (B) in single-threaded version. The max differences were smaller than in the multi-threaded version. The abnormal spiking behavior happened at around 150 epochs, which is also different from the multi-threaded version.

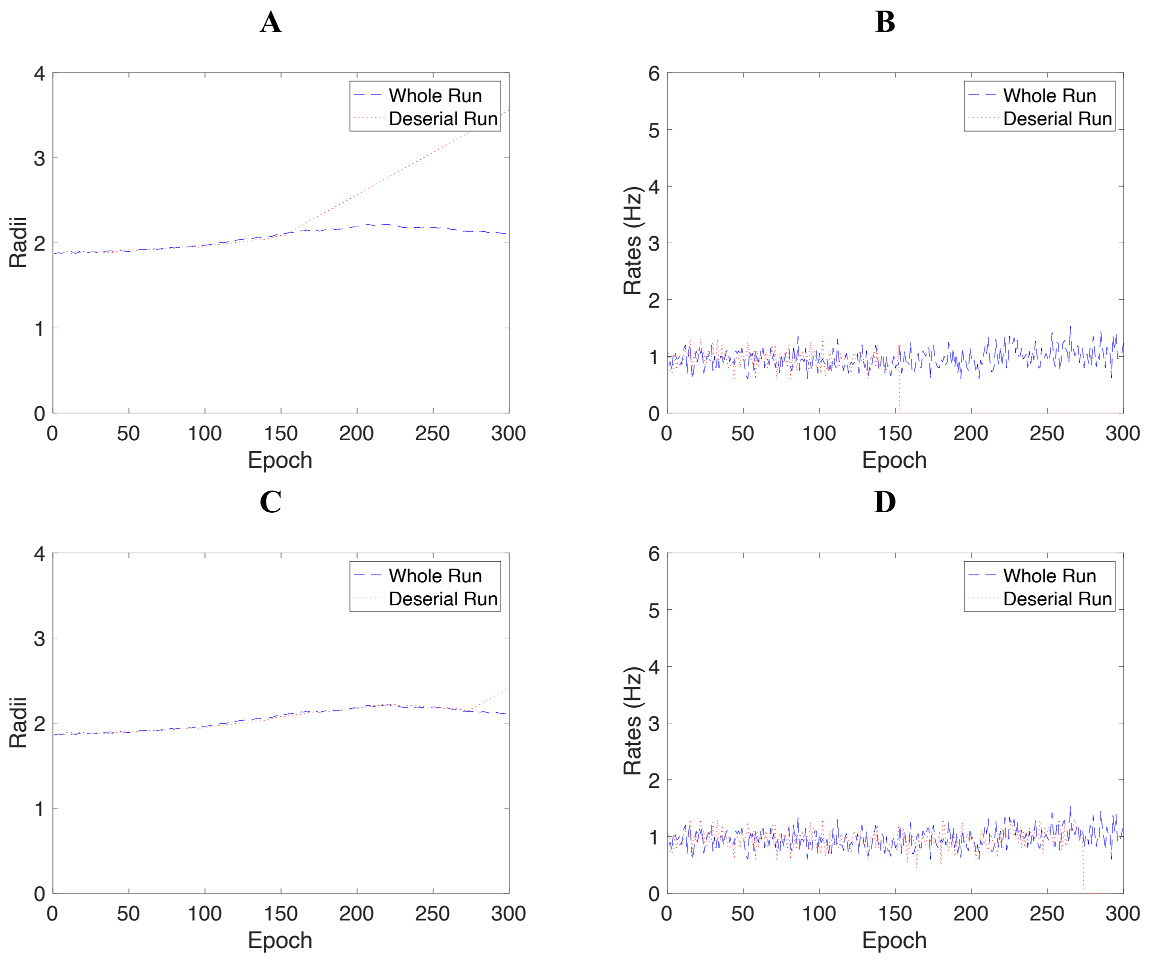
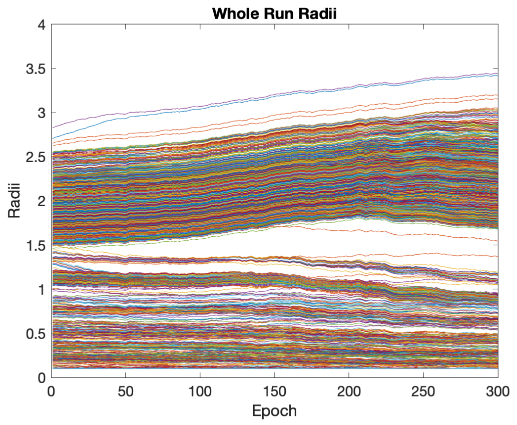


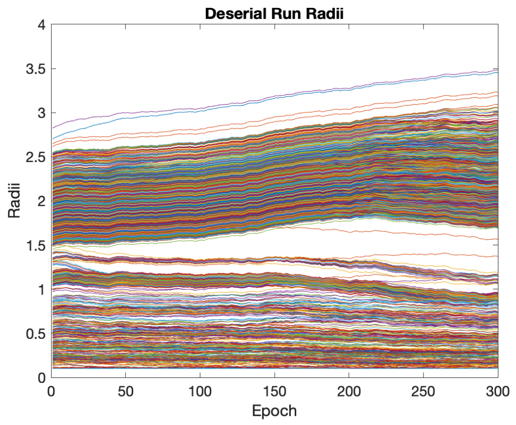
Figure 22: The max radii and rates in each epoch of two neurons exhibited abnormal spiking behavior. The top (A) and (B) are the first neuron's radii and rates, while the bottom (C) and (D) are the second neuron's radii and rates. Both neurons had similar behavior in that they stopped spiking in the middle of the simulation and their radii began to increase.

To identify the cause of this behavior, we further investigated the simulation process by outputting various internal states for each neuron, including the membrane potential and spiking activities in every single time step. We found that the cause of non-spiking was due to calculating an edge case during the growth update. In the growth model, when calculating the overlapping region of two neurons, one very small computed region was generated a non-a-number (NAN) result. This caused the neuron to appear as not spiking during simulation and led to the changes in radii for the rest of simulation. After modifying the code to take this edge case into account, the simulation results showed consistent max differences in each epoch (Figure 23 E, J). This confirmed the error was not from the deserialization implementation. Figure 23 showed the final results after fixing the edge case.

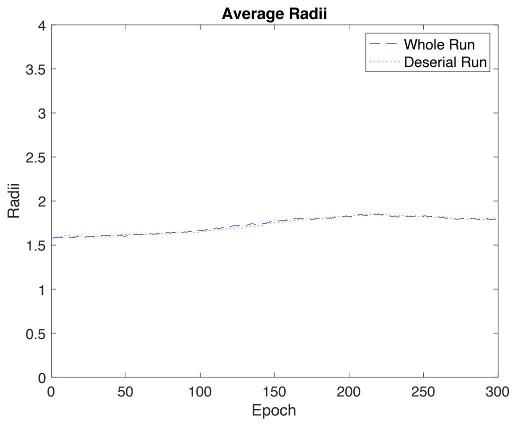
A


Whole Run Radii
Radii
Epoch

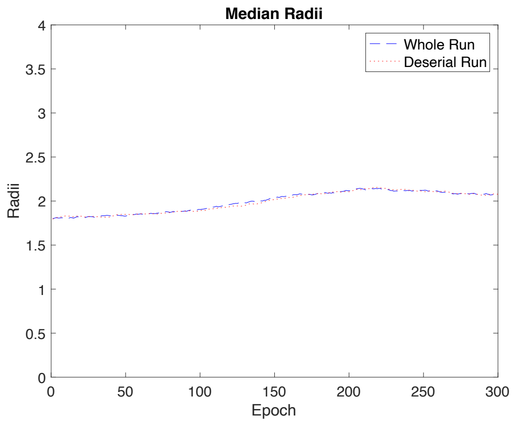
B


Deserial Run Radii
Radii
Epoch

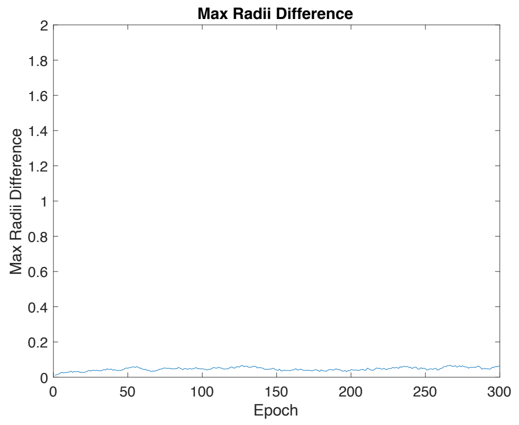
C


Average Radii
Whole Run
Deserial Run
Radii
Epoch

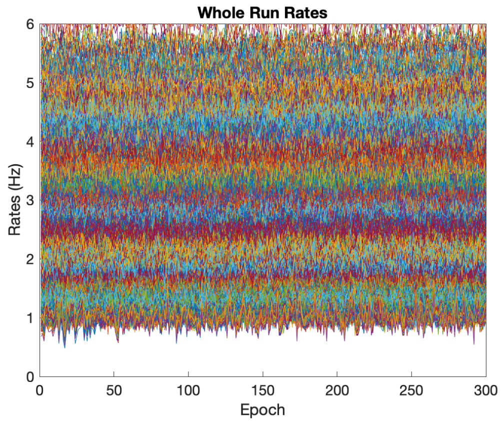
D


Median Radii
Whole Run
Deserial Run
Radii
Epoch

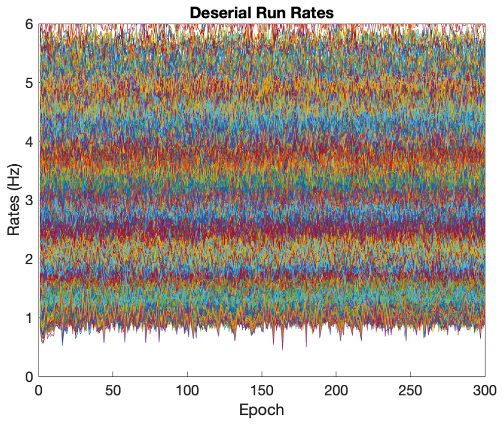
E


Max Radii Difference
Max Radii Difference
Epoch

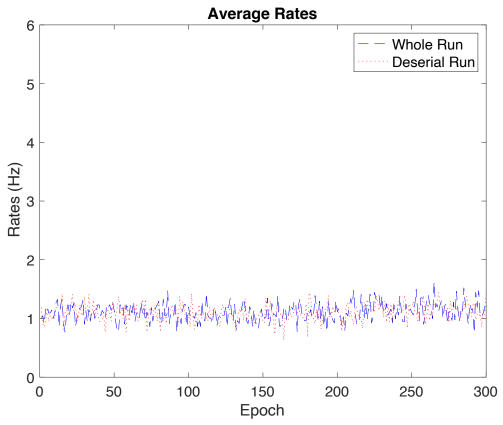
F



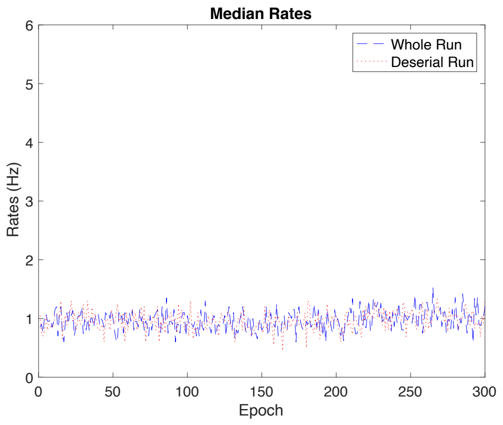
G



H



I



J

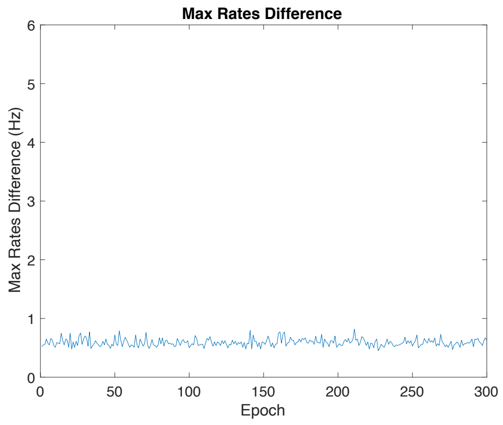


Figure 23: The final results after fixing the edge case error. (A)-(E) are radii from the whole run, the deserialization run, the two runs' average and median radii, and max differences. (F)-(J) are the same but with data in spiking rates.

7.2. STDP Model Verification

The verification test for the STDP model included a CPU-based experiment and a GPU-based experiment. Both experiments used the single-threaded version code. Each experiment followed the STDP simulation workflow described in Figure 7: a growth simulation followed by a STDP simulation. In each experiment, both simulations ran 20 epochs with 900 neurons and the same neuron and layout models were used.

To verify the STDP behavior, we monitored every single synaptic weight adjustment event in both STDP simulations by outputting each spike pair interval and its calculated fractional change in weight from the stdpLearning() method. The calculated fractional changes of weights were plotted against the time intervals and verified with the STDP model in Figure 2. Figure 24 shows the STDP model verification results.

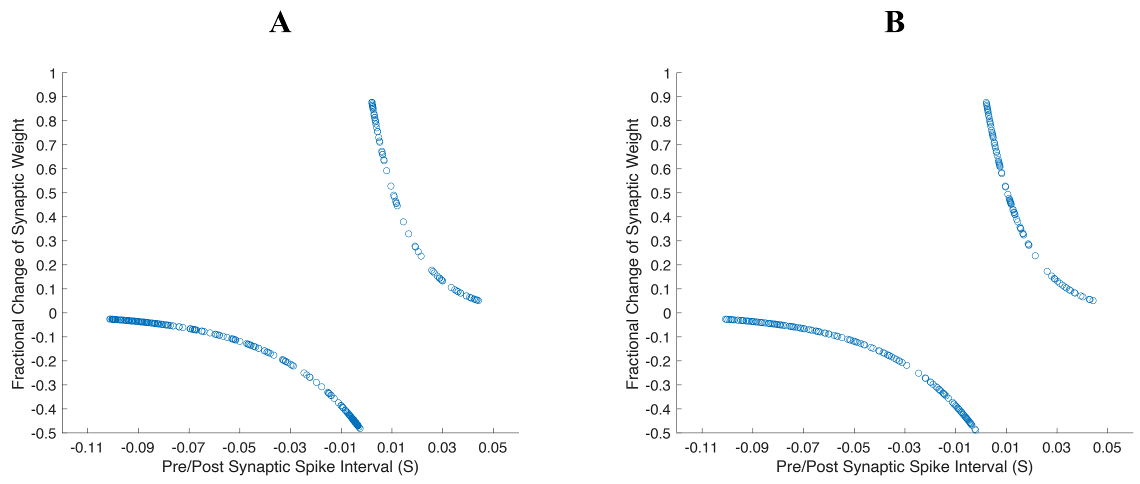


Figure 24: STDP verification test results. The calculated fractional changes of weights were plotted against the spike pair time intervals. (A) is the result of the CPU-based experiment. (B) is the result of the GPU-based experiment.

To understand the overall weight changes after STDP modification, we also serialized the weights after both STDP simulations. We found that both simulations had the same four synapses (out of ~2000 synapses) with changes in weights. These four synapses were two sets of symmetric connections, meaning two synapses had the same two neurons as the source and destination neurons. It was just the direction that was opposite. The other two synapses were the same as the other two neurons. The weight changes of these four synapses were shown in Table 3 and Table 4. Both simulations demonstrated that in each symmetric connection set, one synapse was strengthened to a larger value, whereas the other was weakened to a very small value.

Table 3: The four STDP adjusted synapses in CPU-based experiment

Synapse number	Source Neuron	Destination Neuron	Synaptic Weight Before STDP tuning	Synaptic Weight After STDP tuning
1	1	2	7.35178×10^{-10}	1.40130×10^{-45}
2	2	1	7.35178×10^{-10}	8.45108×10^{-09}
3	3	4	7.46843×10^{-10}	6.61449×10^{-09}
4	4	3	7.46843×10^{-10}	1.40130×10^{-45}

Table 4: The four STDP adjusted synapses in GPU-based experiment

Synapse number	Source Neuron	Destination Neuron	Synaptic Weight Before STDP tuning	Synaptic Weight After STDP tuning
1	1	2	7.36995×10^{-10}	1.40130×10^{-45}
2	2	1	7.36995×10^{-10}	1.40510×10^{-08}
3	3	4	7.37815×10^{-10}	5.02650×10^{-07}
4	4	3	7.37815×10^{-10}	1.40130×10^{-45}

7.2.1. Large-Scale STDP Simulation Demonstration

Finally, to verify and demonstrate STDP in a large-scale simulation, we ran a cortical culture growth simulation followed by 3 steps of STDP tunings. Each step began with deserializing the network from the previous simulation and ended with serializing the network. We monitored the distribution of weights in each step to verify the change in weights. Figure 25 shows the workflow of the large-scale STDP simulation with a growth simulation followed by 3 steps of STDP tuning simulations. The first step ran 20 seconds of tuning, the second step ran another 100 seconds, and the final step ran 200 seconds.

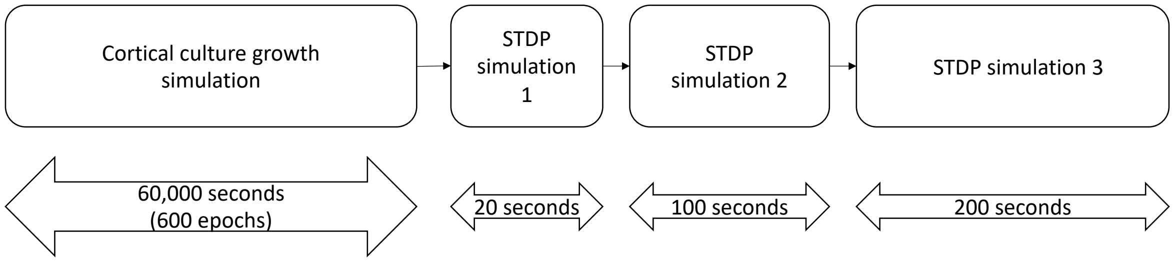


Figure 25: The workflow of large-scale STDP simulation.

Figure 26 showed the synaptic weight histogram in each tuning stage. When comparing the non-STDP tuning and 20 seconds of tuning, about 10% of synapses were strengthened and shifted to a larger value, including 5% that went directly to the maximum allowable weights (the rightmost bar in Figure 26 B). Similarly, another 10% of synapses were weakened and shifted to a value toward zero. When comparing three steps of the tuning process (Figure 26 B, C, D), although they appeared the similar from visualizing the distribution, a small amount of synapses were actually strengthened to the maximum weight value (rightmost bar increased from ~5%, ~7%, and to ~8% in three steps). On the other hand, not much synapses were weakened. These results indicated that at the beginning of tuning, about 20% of synapses were modified rapidly to either increase or decrease their weight values. However, after 20 seconds of tuning, the majority of synapses seemed to be stabilized with only a small number of synapses being strengthened to max.

In addition, the other observation in Figure 26 is the weight distribution shifted from unimodal to bimodal. One hypothesis of the distribution change is a symmetry breaking between those symmetric connections. According to literature, STDP may break this symmetry [26] [27] by causing the weight from a neuron to another become stronger, and the reverse direction weakened. This may be the reason to explain the bimodal distribution.

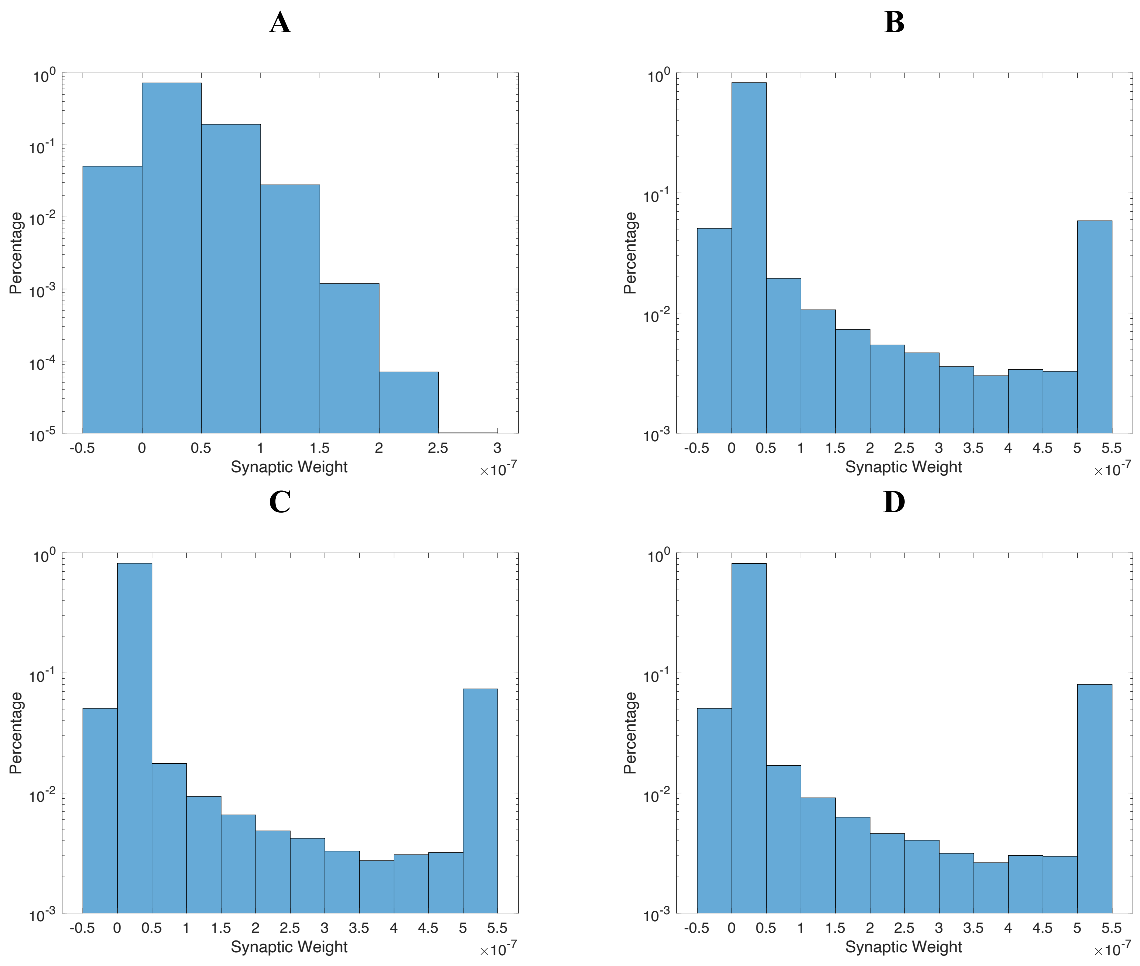


Figure 26: The histogram of synaptic weight distribution in each stage of STDP tuning. (A) is the non-tuning weight distribution. (B) is 20-second tuning. (C) is another 100 seconds tuning. (D) is final 200 seconds tuning. The bin width is set to 0.5 $\times 10^{-7}$.

8. Conclusion and Discussion

In this project, we presented the process of enabling neural network simulations of growth and STDP. Specifically, the design and implementation of the serialization capability is not only useful for STDP simulations but also brings more possibilities to the simulator. For example, serialization can be used in combining other network models to simulate more scenarios. This skeleton design of serialization also provides a foundation for other future applications. In addition, we implemented the basic form of STDP and verified its behaviors. The results showed that the STDP model changed the distribution of synaptic weights by shifting from a unimodal distribution to a bimodal distribution.

What I learned the most from the project is about learning to implement a feature to an existing program. Since it was my first time building a feature on top of a program, understanding the program's structure and architecture became the priority. Although it was very challenging at the beginning to know such a complex program, it really helped me in reviewing other developers' code and understanding their logic throughout the process. Besides, I also learned many new techniques and tools, such as Cereal library, GPU CUDA library, Matlab, etc.

8.1. Future Work

First, given the design of the serialization workflow, we would like to see if serialization can be extended by storing more objects. One possibility could be to use other serialization techniques or modify the existing code to minimize the usage of a raw pointer. For the STDP model, we would like to verify the hypothesis of symmetry breaking and continue to improve the STDP model. This includes adding not only excitatory synapses but also inhibitory synapses, setting up the upper growth limit in synaptic strength, or testing on other maximum synaptic weight values. We also would like to see other types of STDP models implemented.

Bibliography

[1] E. R. Kandel, J. H. (James H. Schwartz, J. Dimes, E. R. Kandel, J. H. (James H. Schwartz, and J. Dimes, *Principles of neural science*, 2nd ed. New York: New York : Elsevier, 1985.
[2] L. Squire, D. Berg, F. E. Bloom, S. du Lac, A. Ghosh, and N. C. Spitzer, *Fundamental Neuroscience*. Academic Press, 2012.
[3] "What are the parts of the nervous system?," *https://www.nichd.nih.gov/*. https://www.nichd.nih.gov/health/topics/neuro/conditioninfo/parts (accessed Apr. 27, 2020).
[4] J. van Pelt, P. S. Wolters, M. A. Corner, W. L. C. Rutten, and G. J. A. Ramakers, "Long-term characterization of firing dynamics of spontaneous bursts in cultured neural networks," *IEEE Transactions on Biomedical Engineering*, vol. 51, no. 11, pp. 2051–2062, Nov. 2004, doi: 10.1109/TBME.2004.827936.
[5] D. E. Feldman, "The Spike-Timing Dependence of Plasticity," *Neuron*, vol. 75, no. 4, pp. 556–571, Aug. 2012, doi: 10.1016/j.neuron.2012.08.001.
[6] H. Markram, W. Gerstner, and P. J. Sjöström, "A History of Spike-Timing-Dependent Plasticity," *Front Synaptic Neurosci*, vol. 3, Aug. 2011, doi: 10.3389/fnsyn.2011.00004.
[7] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann, "Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs," *Science*, vol. 275, no. 5297, pp. 213–215, Jan. 1997, doi: 10.1126/science.275.5297.213.
[8] R. C. Froemke and Y. Dan, "Spike-timing-dependent synaptic modification induced by natural spike trains," *Nature*, vol. 416, no. 6879, pp. 433–438, Mar. 2002, doi: 10.1038/416433a.
[9] D. E. Feldman, "Timing-based LTP and LTD at vertical inputs to layer II/III pyramidal cells in rat barrel cortex," *Neuron*, vol. 27, no. 1, pp. 45–56, Jul. 2000, doi: 10.1016/s0896-6273(00)00008-8.
[10] L. I. Zhang, H. W. Tao, C. E. Holt, W. A. Harris, and M. Poo, "A critical window for cooperation and competition among developing retinotectal synapses," *Nature*, vol. 395, no. 6697, pp. 37–44, Sep. 1998, doi: 10.1038/25665.
[11] G. Bi and M. Poo, "Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type," *J. Neurosci.*, vol. 18, no. 24, pp. 10464–10472, Dec. 1998, doi: 10.1523/JNEUROSCI.18-24-10464.1998.
[12] Y. Babacan and F. Kaçar, "Memristor emulator with spike-timing-dependent-plasticity," *AEU - International Journal of Electronics and Communications*, vol. 73, pp. 16–22, Mar. 2017, doi: 10.1016/j.aeue.2016.12.025.
[13] M. Stiber, F. Kawasaki, D. B. Davis, H. U. Asuncion, J. Y.-H. Lee, and D. Boyer, "BrainGrid+Workbench: High-performance/high-quality neural simulation," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 2469–2476, doi: 10.1109/IJCNN.2017.7966156.
[14] "BrainGrid." https://uwb-biocomputing.github.io/BrainGrid/ (accessed May 08, 2020).
[15] F. Kawasaki and M. Stiber, "A simple model of cortical culture growth: burst property dependence on network composition and activity," *Biol Cybern*, vol. 108, no. 4, pp. 423–443, Aug. 2014, doi: 10.1007/s00422-014-0611-9.
[16] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input," *Biol Cybern*, vol. 95, no. 1, pp. 1–19, Jul. 2006, doi: 10.1007/s00422-006-0068-6.
[17] H. Markram, Y. Wang, and M. Tsodyks, "Differential signaling via the same axon of

neocortical pyramidal neurons," *PNAS*, vol. 95, no. 9, pp. 5323–5328, Apr. 1998, doi: 10.1073/pnas.95.9.5323.
[18] S. L. Jackman and W. G. Regehr, "The Mechanisms and Functions of Synaptic Facilitation," *Neuron*, vol. 94, no. 3, pp. 447–464, May 2017, doi: 10.1016/j.neuron.2017.02.047.
[19] A. Van Ooyen, J. Van Pelt, and M. A. Corner, "Implications of activity dependent neurite outgrowth for neuronal morphology and network development," *Journal of Theoretical Biology*, vol. 172, no. 1, pp. 63–82, Jan. 1995, doi: 10.1006/jtbi.1995.0005.
[20] F. Kawasaki, "Accelerating large-scale simulations of cortical neuronal network development," Thesis, 2012.
[21] "Java Object Serialization." https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html (accessed May 18, 2020).
[22] "Boost C++ Libraries." https://www.boost.org/ (accessed May 18, 2020).
[23] "cereal Docs - Main." https://uscilab.github.io/cereal/index.html (accessed May 08, 2020).
[24] "Protocol Buffers," *Google Developers*. https://developers.google.com/protocol-buffers (accessed May 18, 2020).
[25] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 1999.
[26] M. Gilson, A. N. Burkitt, D. B. Grayden, D. A. Thomas, and J. L. van Hemmen, "Emergence of network structure due to spike-timing-dependent plasticity in recurrent neuronal networks. II. Input selectivity--symmetry breaking," *Biol Cybern*, vol. 101, no. 2, pp. 103–114, Aug. 2009, doi: 10.1007/s00422-009-0320-y.
[27] C.-W. Shin and S. Kim, "Synaptic symmetry breaking by spike timing dependent synaptic plasticity," *BMC Neuroscience*, vol. 9, no. 1, p. P100, Jul. 2008, doi: 10.1186/1471-2202-9-S1-P100.