MicroNSL Module – Neuron Spiking Simulation in Java

Eduardo Zuloaga
M. Llofriu, A. Weitzenfeld

Abstract

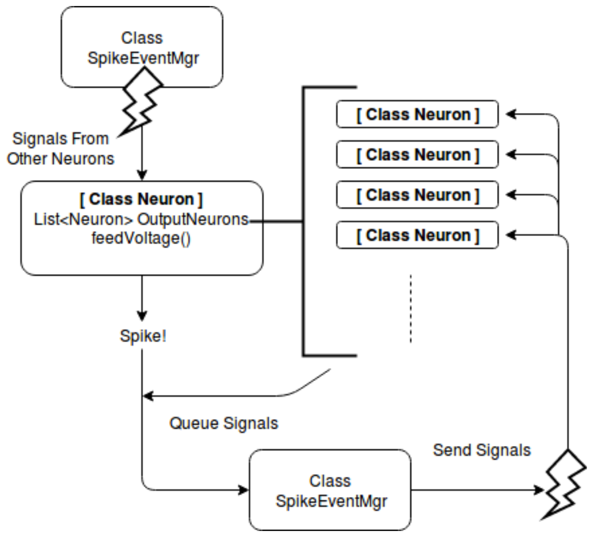
Neurons in the brain exhibit a specific spiking behavior in transmitting impulses between one another. When a Neuron has received enough electrical charge to reach a predetermined membrane potential, the neuron will fire and send charge to all neurons connected to its axons. When we simulate a network and observe this behavior within a computer, we're able to see some interesting patterns occur.

1. Introduction

The idea behind simulating a basic spiking neural network (SNN) in a program is simple at surface level: we create a Neuron object, instantiate a number of them, and have them send charges to each other while reading their membrane potentials and keeping track of when they spike. Because these events occur as in the same domain of space-time, the entire simulation space must be kept synchronous, thus we will need to orchestrate the simulation via an event manager. At each time step the event manager will process all spikes, queue any new ones, and advance the simulation.

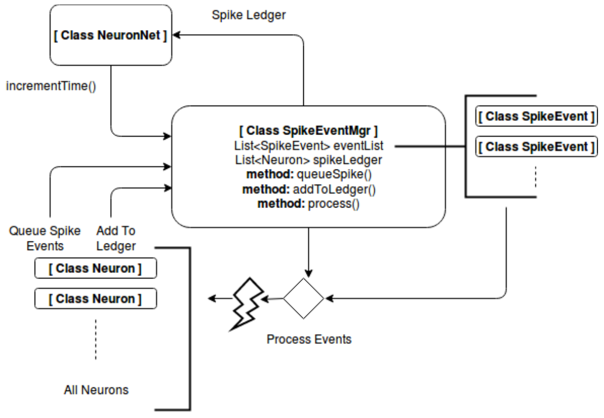
2. Java Classes

At the most basic level, the simulation will consist of the Java classes NeuronSpike, the main driver class, SpikeEventMgr, our singleton event manager, and Neuron, which will receive some voltages, spike, and send some voltages. The Neuron class is the foundational first-class citizen of our project. How it works is as follows: our Neuron will be connected to a number of other output Neurons and keep a ledger of its connections in a List<Neuron> object. It will receive a number of voltage signals from other Neurons, orchestrated by the SpikeEventMgr instance. If its voltage surpasses a predefined threshold, the Neuron will spike and tell the SpikeEventMgr instance to deliver a signal to the output Neurons at the correct time step. The diagram below describes the process visually.



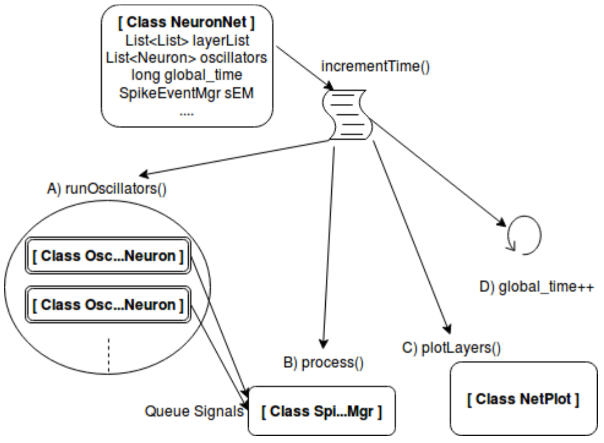
The Neuron class was, however, abstracted and split into two different subclasses, GenericNeuron and OscillatorNeuron, for reasons we will discuss momentarily.

The SpikeEventMgr class exists to consume and queue requests for voltage transmission from Neuron to Neuron at an arbitrary time step, as well as keep a ledger of all spike events that occur for plotting purposes. SpikeEventMgr maintains all queued spike events in a List<SpikeEvent> object. Amidst potentially confusing naming conventions it is imperative to understand what a "spike event" is in the context of the event manager: it is an event where a Neuron is to receive an impulse at a given time, *not* where a Neuron is meant to spike at that time. Every element in the SpikeEvent list contains data pertaining to the target Neuron, when the transmission should be received, and the magnitude of the signal. When the simulation advances to a new time step, SpikeEventMgr will traverse its spike event list and carry out all events designated for that time. In this way we keep the simulation synchronous. This model functions well under the assumption that all spike events are queued for times after the current time, lest we find ourselves time traveling.



Generating some Neuron objects manually and sending a charge to one of them was enough to give us proof of concept, but in order to see anything meaningful we need to procedurally generate an SNN. This task can be accomplished by generating the network by layers – each layer houses a certain number of Neuron instances which are each connected to *some* Neuron instances in the previous layer. The NeuronNet class exists for this purpose.

NeuronNet will take in as constructor parameters: the number of layers, an array of integers in which each index represents a layer and each element represents the number of Neurons to be generated, an array of integers to describe connectivity between each layer to the one previous, and some miscellaneous data pertaining to Neuron properties. Using this information, it will generate all the required Neurons and group them into layers, all of which it will store in a List object called layerList, and make axon connections between them based on connectivity parameters. NeuronNet also houses the capability to manage instances of a special Neuron subclass called OscillatorNeuron, which exists only to externally inject voltage into the generated network and can be added and removed at will when the simulation runs.

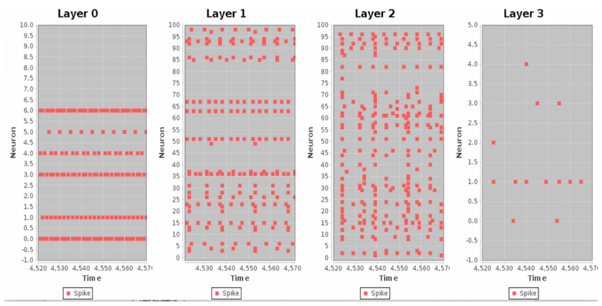


For this purpose the Neuron class is abstracted into two subclasses: GenericNeuron and OscillatorNeuron. GenericNeuron is to be used as a procedurally generated Neuron and is considered part of the SNN. The OscillatorNeuron is generated and used after the simulation begins, sitting on a non-existent "meta" layer feeding voltage to a single GenericNeuron. There are three types of Oscillator types: "Continuous", spiking at fixed user-defined intervals, "Probabilistic", having a defined probability to spike at each time step, and "Sine", which works similar to the Continuous class under the hood but oscillates its spike intervals on a sine wave with a user-specified wavelength.

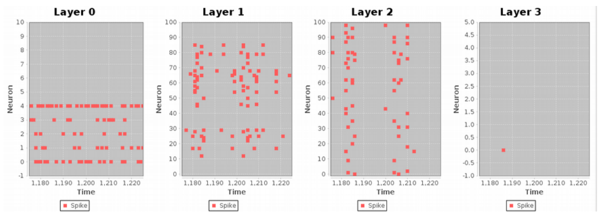
The NetPlot class uses the spike ledger from SpikeEventMgr to plot out layers and their spiking neurons on the screen. The ControlPanel class spawns a GUI from which oscillators can be added and removed.

3. Observations

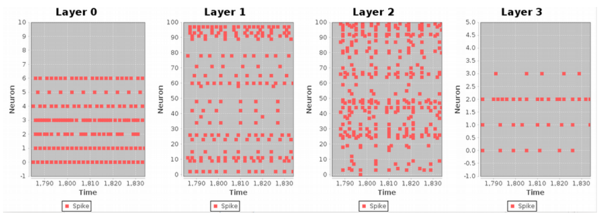
The behavior of the network was observed in a variety of different contexts to see if they yielded any interesting spiking patterns. As it stands all connections between Neurons in the network share a common delay (1 time step) and connection weights between 0.1 and 0.3 are generated per connection. With different assortments of my 3 oscillator types: Continuous, Probabilistic and Sine, the network demonstrates various behaviors.



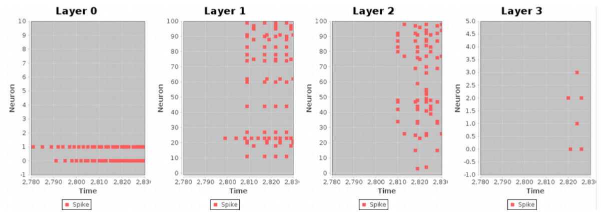
Strictly inserting oscillators of a Continuous type generates repetitious, deterministic spiking patterns in the network. Wavelengths of these repetitions vary with differing generation of the network as well as the spike frequency of the input Neurons. Another observation I made was that this spike pattern wavelength in a layer was either equal to or greater than that of the layer before it. While some neurons are able to spike a greater frequency than those in the previous layer, all neurons in the layer will never exhibit collectively repetitive behavior at a rate greater than a previous layer following similar behavior. **Figure 1** illustrates an example of this behavior, in which the wavelength of the repetitious patterns in layers 2 and 3 were too large for our capture window, but is plainly displayed in layer 1.



Using strictly Probabilistic oscillators did away with the deterministic behavior of the network and introduced (pseudo) randomness. Probabilistic oscillators follow no fixed patterns so we can see no such behavior in later layers. With a single oscillator, it can be observed that all neurons connected to it spike uniformly, however, this occurs across irregular intervals. When a higher number of Probabilistic oscillators are added to the network, spiking patterns become much less discernible and eventually nonexistent. Oscillators of higher probability excite the network further. **Figure 2** shows one such example of several probabilistic oscillators feeding the network, with some aforementioned "uniform" spiking occurring in Layer 2 and the occasional spike in Layer 3.



Using a mixture of Continuous and Probabilistic oscillators yielded a hybrid of the two previous behaviors. With a high ratio of Continuous to Probabilistic types, the network exhibits repetitious behavior as observed using purely Continuous oscillators, however, with a degree of randomness. The fixed interval behaviors are present, skewed by some excited Neurons spiking earlier or later than expected. **Figure 3** demonstrates this skew. Note the uniformity of the spiking Neurons occurring in Layer 1, with some outliers spiking before or after the expected time. As voltage propagates further into the network, we see the effect of the Probabilistic oscillators gradually increase, and uniform intervals start to disappear by the time our signals reach Layer 2. As we increase the number of Probabilistic oscillators, noise throughout the network is greatly amplified. A noteworthy observation is that Continuous-type oscillators contribute much less to the uniformity of the network than Probabilistic-type oscillators contribute chaotic behavior. To observe "semi"-uniform behavior, then, one must use very little Probabilistic-type oscillators and a great number of Continuous-type oscillators.



Using strictly Sine-type oscillators introduces a unique type of behavior: when the sine waves are synchronized, the rest of the network spikes in a highly uniform manner. Areas where the sine waves are not in sync produce noisier, harder to predict results. An interesting and altogether not unexpected observation was that if many oscillators of the same wavelength are used, the rest of the network spikes in a repetitive manner of a similar (although not exactly the same) wavelength as the oscillators introduced. **Figure 4** illustrates the beginning of such an oscillation. The "sameness" of this wavelength is determined by how large a wavelength is allowed by the user. If wavelengths are smaller, then Neurons are given less time to decay their potentials and will spike more. Because they can only spike once per time step no matter how much voltage is fed to them, this can result in a great deal of energy lost in the network. When the voltage is introduced in oscillators throughout a much larger margin of time (as it is with Sine-type oscillators of a large wavelength), less "information" is lost and the Neurons follow a pattern more closely defined as a function of that wavelength. Using many Sine waves of differing wavelengths will, in theory, create repetitious behavior as their spike rates synchronize, but this will occur at larger intervals than we care to observe if so, with some deviation.

4. Potential Improvements

The current network model, while it works enough to make plain observations about its behavior, is clearly not enough to be reflective of a real neural network, nor is it something that provides a reasonable degree of utility.

Firstly, the entire network's generation is static from the moment the simulation is started: Neurons are generated, numbers are generated from seeds, connections are made, and the simulation runs. In the future the ability to visualize, rearrange, and alter connections between neurons would prove useful when trying to examine specific behaviors. Additionally, functionality for the network to generate feedback loops between Neurons would prove useful, in contrast to the current pure feed-forward model. Doing this would allow us to observe charges that bounce around between neurons and linger within the network well after it has been excited. Neurons with regressive signals could also be introduced, negating voltage to a connected neuron when they spike instead of adding to it. Delay of connections between neurons is also, as it stands, declared globally and goes unchanged from the start of the simulation to the end. A useful feature in the future would be to implement a manner in which we can generated connection delays in a meaningful way.

Additionally. the spike ledger functionality of the SpikeEventMgr class, in this case, lends itself to metrics collection. Information about the entire network, such as how many neurons spiked at a given time step in each layer, the mean amounts of spikes per minute and standard deviation, or ways the current network configuration could be improved are potential metrics that could be collected and analyzed to grow the project further.

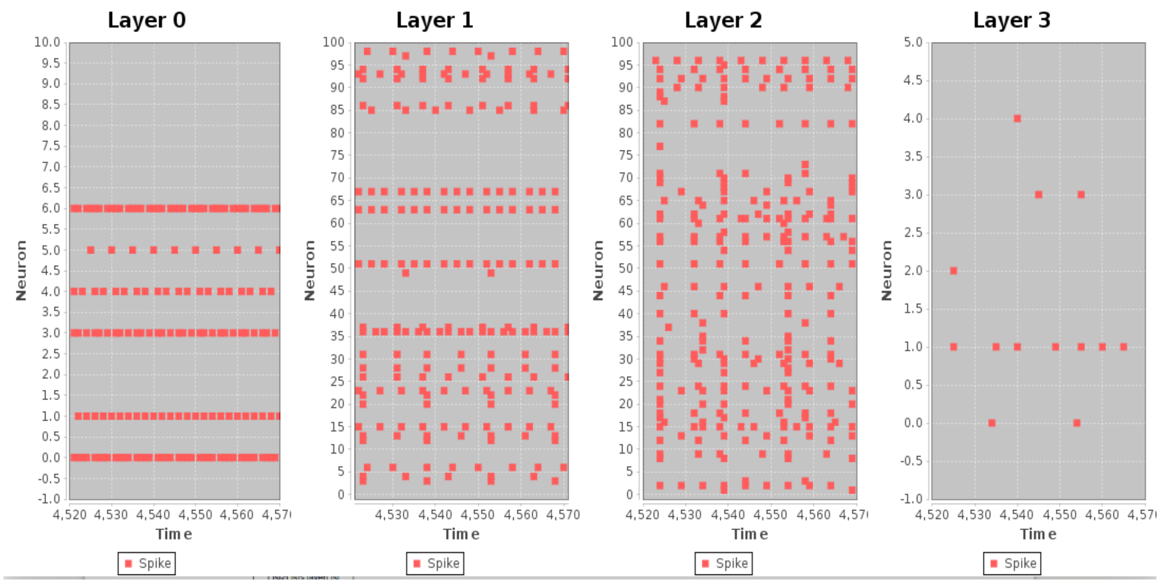
5. Challenges

A framework for modeling and simulating an SNN has been created and tested, one in which we have insofar been able to gather meaningful observations. I've learned a great deal about constructing a synchronous event-based simulation and have had some pitfalls, firstly in the area of performance. When I set up the framework for the project and tested it with a network consisting of hundreds of neurons across several layers, I noticed the simulation ran much too slowly to make observations in a reasonable amount of time. It was only after using complexity-efficient data structures that I was able to significantly cut down on execution time, particularly in my Neuron lists, which were previously composed of linked lists whose elements took a maximum O(n) complexity to dereference. Since then I placed a great amount of focus on writing and reading data in a computationally effective way, which has helped scale the project.

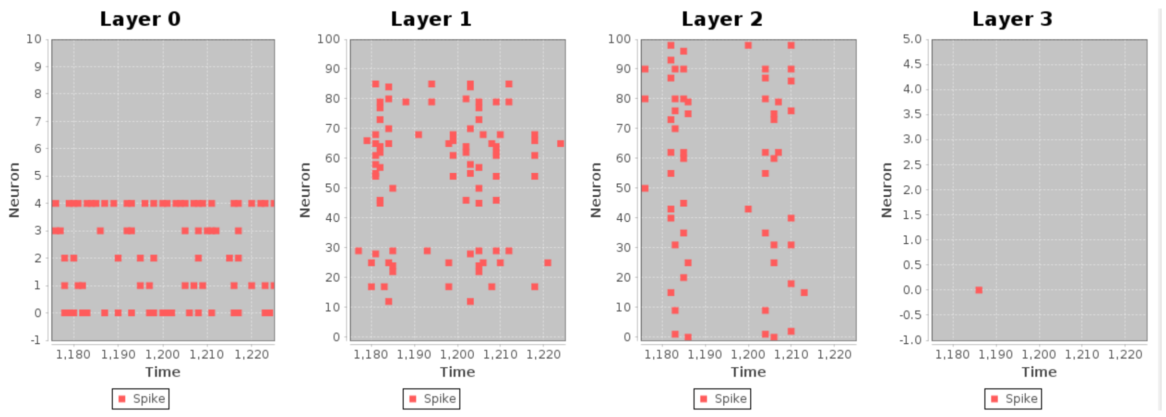
My other large hurdle to jump was that of plotting my spikes to easily observe the network. In the very beginning JChart2D was used, however I discovered that this provided little functionality for scatter-plots. Another library, JfreeChart, looked promising as it boasted native scatter-plot capability. JfreeChart is the library I ended up using and was able to implement with great success. Out of this arose the necessity for a data structure that housed references to all of the spiked neurons per time step, and thus this functionality was added to the SpikeEventMgr class.

6. Conclusion

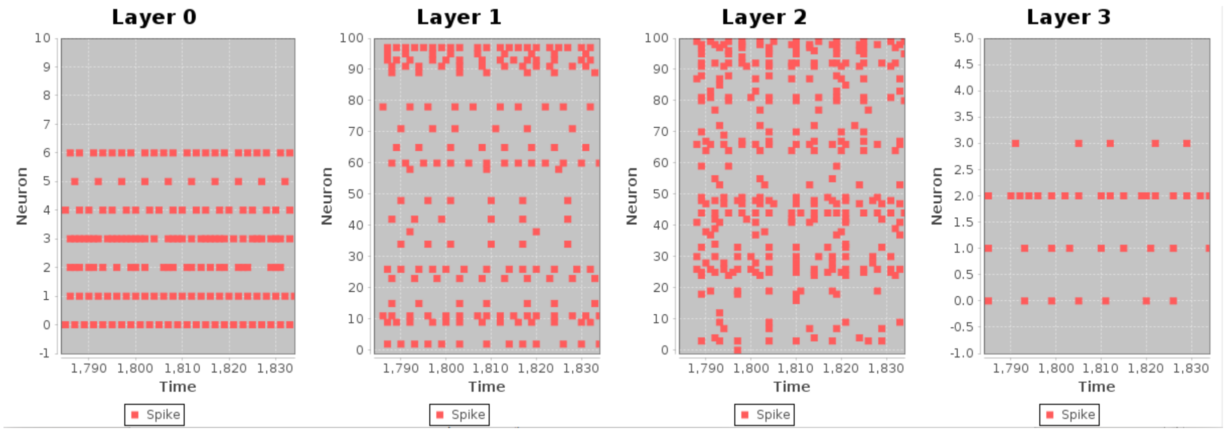
The NeuronSpike project is quite a ways from accurately emulating a true spike-driven neural network, but many patterns can be observed in its current iteration and it provides the framework for a key component of MicroNSL. May it provide great insight in the near future and serve as one of many valuable stepping stones for simulated neural network research.



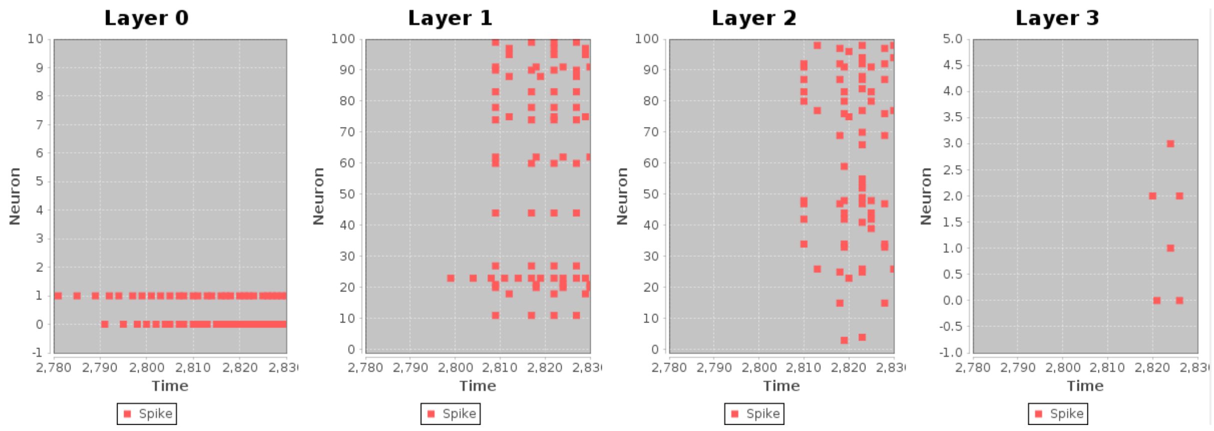
* **Figure 1: Continuous Oscillators**



* **Figure 2: Probabilistic Oscillators**



*** Figure 3: Probabilistic, Continuous combined**



*** Figure 4: Sine Wave Oscillators**