Charles J. Simon's Brain Simulator II

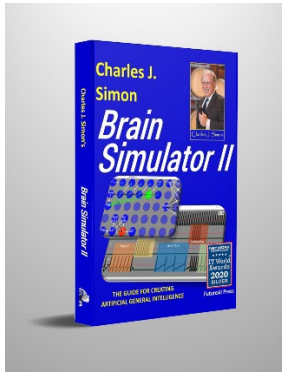
- What makes the Brain Simulator Unique? Biological Plausibility. Several spiking neuron models are implemented based on biological neurons.
- The User Interface. A graphical display of neurons so users can explore the internal workings of the network in real time.
- Software Modules. Modules allow any cluster of neurons to have any function defined by software.
- Multiple Data Types. Any AGI must integrate diverse information coming from multiple senses—sight, sound, touch, and time.

THE MANUAL FOR CREATING ARTIFICIAL GENERAL INTELLIGENCE

NETWORK PRODUCTS GUIDE ★★★★★ IT World Awards 2020 SILVER

FutureAI Press

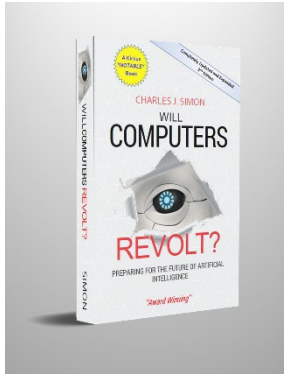
Books by Charles Simon



Brain Simulator II: THE GUIDE FOR CREATING ARTIFICIAL GENERAL INTELLIGENCE

The companion book to the Brain Simulator II software, it contains everything you need to get started experimenting with Artificial General Intelligence (AGI). This book includes descriptions of several spiking neuron models, the User Interface, the Neuron Engine, and Software Modules for functioning neuron clusters, and AGI applications.

172 pp. 2021



Will Computers Revolt? PREPARING FOR THE FUTURE OF ARTIFICIAL INTELLIGENCE

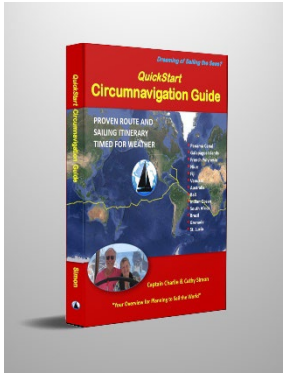
This award winning and well-reviewed book, Describes the When? Why? and How Dangerous? of future computers which will exceed human abilities. It is not all doom and gloom, but there are actions we should be taking now!

2nd Edition, Updated and Expanded, 360 pp. Coming in May 2021

Computer Aided Design of Printed Circuits: THE GUIDE FOR EVALUATING, PURCHASING, AND USING COMPUTER AIDED DESIGN SYSTEMS

From defining what printed circuits are, to how a computerized printed circuit design system works, to explaining the hardware and software of the system, and acquiring and using the design system, this book will give the reader a complete understanding of the process.

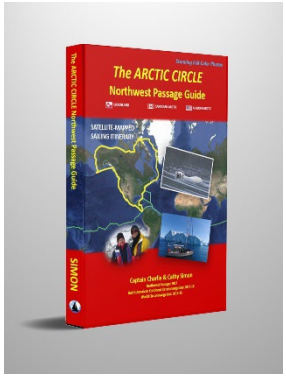
357 pp. 1987, Currently out of print.



QuickStart **Circumnavigation Guide:** PROVEN ROUTE AND SAILING ITINERARY TIMED FOR WEATHER

Dreaming about Sailing the Seas? Get ready to get off the dock and sail YOUR OWN WORLD CRUISE! In this adventure of a lifetime, Capt. Charlie and Cathy Simon spend 14 months visiting five continents, 16 countries and crossing three major oceans, plus, many Seas sailing a 26,000-mile circumnavigation in 2014-15. It is easy to read, well organized, and entertaining.

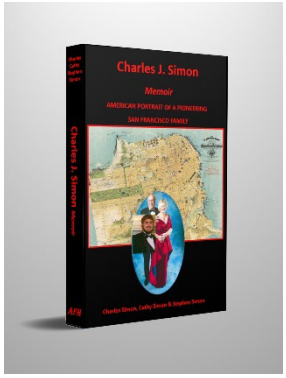
166 pp. 2016



The ARCTIC CIRCLE Northwest Passage Guide**:** SATELLITE- MAPPED SAILING ITINERARY

This second book in the World Sailing Guru series follows the adventures in the ice of world circumnavigators Captain Charlie and Cathy Simon as they sail their way with their crew of world sailors through the legendary Northwest Passage of the Canadian Arctic and Alaskan Arctic in 2017. Returning to the US East Coast via the iconic Panama Canal the book includes their passage notes of the circumnavigation of the North American Continent in 2018.

204 pp. 2020



Charles J. Simon, Memoir: AMERICAN PORTRAIT OF A PIONEERING SAN FRANCISCO FAMILY

Beginning in the 1800's the Simons and the Schoenfelds made their fortunes in and around San Francisco, California. From Civil War restrictions, to taking passage on the first transcontinental railroad train, the Great 1906 earthquake and fire, the Panama–Pacific International Exposition in 1915 celebrating the completion of the Panama Canal, to today's technology endeavors my family has played a part. The book showcases spectacular San Francisco events.

145 pp. Coming in May 2021

Also by Charles Simon

Visit https://futureai.guru/founder.aspx for a complete publication list.

Software/Hardware, Charles Simon

The Brain Simulator II

The BRAIN Simulator: Tutorial Software for Neural Circuit Design

EEG System (Brainwave Monitoring)

EMG EP, neurodiagnostic software

Synthetic Intelligence

Cynthia Voice-activated Intercom

3-D ComputerScape

3-D MiniCAD for Windows

3-D Mouse

Continuum: Software for Enterprise CAD

Printed Circuit CAD Graphics

Committee Boat Suite (Software for Sail Racing Support)

Flying Media: Museum Interactive System

Passport to Discovery: Museum Interactive System

BRAIN SIMULATOR II

THE GUIDE FOR CREATING ARTIFICIAL GENERAL INTELLIGENCE

CHARLES J. SIMON

FutureAI Press
Washington, DC

http://brainsim.org

Published, April 21, 2021, in the United States by FutureAI Press,
455 Massachusetts Ave NW #120, Washington, DC, 20001, info@futureAI.guru

ISBN-13 (eBook): [TBD]
ISBN-13 (Paper): 978-1-7326872-4-0
Printing Version: 9/25/2023
First Edition

Book Sales, worldwide through Amazon.

Some of the images in this book are available for use under various Creative Commons licenses. These licenses require that URL links to the license text accompany the use of the photograph. For reference, the license URLs are as follows:

CC BY 1.0	https://creativecommons.org/licenses/by/1.0
CC BY 3.0	https://creativecommons.org/licenses/by/3.0/
CC BY-SA 2.0	https://creativecommons.org/licenses/by-sa/2.0
CC BY-SA 3.0	https://creativecommons.org/licenses/by-sa/3.0
CC BY-SA 4.0	https://creativecommons.org/licenses/by-sa/4.0

Table of Contents

Preface:
Computers I Have Known

When I was a senior in high school, one of the math teachers wangled a gifted-students grant and ran an after-school class in computer programming. He'd taken a course the previous semester so had at least a few months' head start on his students, there were, perhaps, a dozen of us. The grant included an allotment of computer time on the University's new IBM 360. Computer time was charged by the second and we were allowed to submit card-decks which would be run overnight and receive printouts in the morning. The teacher would do this once a week…consider the impossibility waiting a week for a typo to be flagged so you could correct it and resubmit the deck.

Also, in my senior year, I was part of a program which allowed me to take one class at the University. While most of my cohorts in the program were taking Western Civ, I signed up for Physics 4 (for majors). Since I was driving out to the University four days a week, and my father's faculty parking permit was clear across the campus from the physics lecture hall, I was nominated to detour through the basement computer facility and deliver the cards and pick up the printouts whenever I could. This also gave me the ability to spend more time punching up program card decks.

At that time, the language of choice for scientific programming was FORTRAN while for business it was COBOL—we only learned FORTRAN. There was a student-version of FORTRAN from the University of Waterloo called WATFOR which ran on the 360 and kept us on the straight and narrow. The more generic FORTRAN IV was more powerful but less forgiving, powerful enough that little errors could require a complete system restart. It wasn't too long

before one of the enterprising students realized that he could insert Job Control Language cards into the program deck which would terminate WATFOR and fire up FORTRAN IV and then request tapes to be loaded or get access to disk drives and on and on, and the computer operator would dutifully comply because they didn't really have a way of knowing where the requests were coming from.

After these shenanigans, we were on the lookout for alternative computer resources and we discovered the IBM 1130 in the physics department. Time on that computer wasn't charged anywhere as far as we could tell and as long as advanced physics students weren't around, we could do whatever we liked.

The school district had an IBM 1401 which was used to process grades etc. and we had a tour and got the manager to let us use it in slack time. It was a few refrigerator-sized cabinets and had an add-on washing-machine which had an additional 4K of memory. It had a FORTRAN II compiler which was so slow we only used it once. It took 15 minutes to compile one of our simple card-decks.

My success in the physics class allowed me to transfer to UC Davis which, at the time, was the most popular campus in the popular UC system. The main computer there was a Burroughs 6700 which had Remote Access! You didn't have to work with card decks, you could write your program at a terminal and it might run only a few minutes later. With each class which required computer work, I'd get an account with a limited amount of computer time. As I was never meticulous enough to get programs to run the first time, once again, I had to hunt up addition computer resources.

The Electrical Engineering Department had a Xerox Sigma 7 computer and the department chair was using it to build an array processor, the forerunner of todays GPU graphics chips. Once again, if I was willing to work around everyone else's schedule, I could use the machine as much as I liked. I recall one of the early assignments was a version of the now-famous Travelling Salesman problem with the objective of finding the most efficient route for visiting a number of destinations. The problem is now famous because it is an example of an intractable problem where the required number of computations goes up as some power of the number of candidate destinations.

The Sigma computer lab was shared by two analog computers. These were programmed by plugging wires into a patch panel and could solve some classes of differential equations among other things. Within a few years, analog computers were completely obsolete because it became possible to simulate the analog computer faster and more accurately on a digital computer.

Midway through my first quarter at Davis, my roommate struck gold (or perhaps copper). A few years earlier, all of the telephone switching systems in the dorms had been replaced and the racks of cast-off telephone relays were languishing in a barn (Davis had a big Ag school) where they hadn't been able to interest a scrap dealer to recycle them. We took the relays over to Electrical Engineering and got independent study credit for building a computer out of them. After months of soldering, the thing clattered away and could add and subtract in a rack about the size of a tall bookcase.

The following year, we acquired a teletype and a paper-tape reader/punch and interfaced them so the relay computer could clatter away in two desk-sized racks and output messages. The problem was that the teletype needed pulses about 8ms long while the telephone relays weren't able to do anything in less than 12ms. To make an 8 ms pulse with 12 ms relays, you can't speed them up but you can slow them down. You start the pulse with a 12 ms relay and end it with on which has been slowed down to 20 ms. Turns out that you can make arbitrarily short pulses with arbitrarily slow devices—but it takes a lot of relays.

Now, time-travel into the present where I'm studying how the human brain works and speculating on why over 65% of your brain is involved in controlling your body while less than 20% is involved in thinking. The answer is the same as for the telephone relays, neurons are slow relative to the signals your body needs for quick actions. So, it takes a lot of neurons.

In my senior year of college, I took a graduate seminar in computer graphics and created a drawing system as the class project but computer resources were still an issue. We were given a small account for the course but it wasn't nearly enough. On the upside, to encourage general student computer use, every student was given a $10 account to access the central computer. Many students had no interest so I went down the hallway in the dorm, knocking on doors,

and getting friends and neighbors to give me the username and password for their gratis accounts. $10 wasn't enough to accomplish much but there was a quirk in the Burroughs operating system; it only checked your available account balance when you logged in. So armed with a page of accounts, I could go to the terminal room, log in, and use a terminal until I got too sleepy to continue. If the central system crashed, I'd need to get out a fresh account because each $10 account would burn through in less than an hour of connect time. I got an A on the project.

When I graduated, I had plenty of job offers because EE grads were in short supply. Said I to myself, "I can get a job any time," so I turned them all down and started my own company to do CAD. Even minicomputers were expensive and the idea was that if the graphics were performed on a workstation microprocessor, the minicomputer could be shared among many graphics terminals and still give snappy performance.

Graphics terminals with embedded processors were beastly expensive ($20K) but I could buy the graphics display for $3K and build my own processor. The first prototype was built on an IMSAI kit with an 8080 CPU programmed in Assembly Language. That CPU could add and move data around but if you needed to multiply, you needed to write your own program. Graphics displays require a lot of multiplication.

For storage, in addition to 32K of RAM, I interfaced a floppy disk which I could do because I'd been freelancing with a company which made disk controllers and also the custom 8X300 microcomputer which controlled them. The floppy drives were about the same speed and capacity as the hard drive on the IBM 1130 of five years earlier.

For a central computer, we leased an Interdata 7/32 which was a clone of the IBM 360. Interdata was chosen initially because they had a compiler which would allow multiple users to run the same program without having multiple copies of the entire program. This is the same idea used by today's .DLL which every modern program uses. As we entered into various marketing arrangements with computer vendors, out computer room filled with a variety of additional computers: Perkin Elmer, A DEC VAX/750, and computer from Harris and others.

On the microprocessor front, IBM came out with the original PC and we bought one for home. It had dual floppy drives, 640K of RAM and a monochrome screen with a green phosphor. These were 5¼" floppies while we were still using the 7" floppies for the smart graphics terminals at work. The lack of a hard drive made the machine close to unusable, so I eventually replaced one of the floppy drives with a 5Mb hard drive.

After this, there came a series of progressively more powerful desktop machines. In 1988, I got the FORTRAN compiler for MS-DOS and wrote the original *Brain Simulator*. It worked within the 640K RAM limit and so would only support 1,200 neurons as compared with the current implementation which supports billions. It could process the neuron array a few times per second which is at least within a few orders of magnitude of the speed of the biological neuron which and spike 250 times per second.

Working on my master's degree, I took a course in parallel processing. I don't recall the brand of computer, but it had 16 PC-equivalent processor boards. I implemented the *Brain Simulator* and a few other programs in parallel, experience which has stood me in good stead to this day. The *Brain Simulator II* works seamlessly across the multiple cores of today's CPUs and can extend via a network to any number of physical machines.

Windows 3 came out and I got a copy. It was unreliable and slow but sometimes worked. Then I got a contract in Silicon Valley working on a Windows application and it completely changed my point of view. Instead of being concerned about the unreliability, I was impressed than anything worked at all. The early Windows releases were based on the assumption that applications would be cooperative and well-behaved. Of course, programs are NOT bug-free, and the slightest problem in any app could bring down the entire system, leading to the infamous "Blue Screen of Death." When Windows NT was developed, it was managed by the architect of the VAX/750 OS, VMS, which was way ahead of its time, and he knew what he was doing so NT was a reasonably stable and reliable system. Much of it survives in today's versions of Windows.

Windows 95 was the last of the non-NT lineage and I was called into Microsoft to help with applications to give Windows 95 users something useful to do. This was concurrent with the emergence of

the Internet and Microsoft figured out, at the last minute, that content and a browser were as important as an application on a local machine. So, they invented MSN, the Microsoft Network, to host a number of useful applications. These included Expedia and the news site I worked on which was started as MSN News, then became a joint venture with NBC dubbed MSNBC, and is now the website of NBC News. Over the two years I worked there, it was the largest news site on the web and we celebrated when we exceeded a million unique visitors in a day.

Large websites like MSNBC.com are distributed across an array of servers and the interesting technical problem was how can you build a large site across numerous servers and not break all the page links whenever you updated the site because the updates would necessarily arrive at different servers at slightly different times. We wanted to update the site any time whenever a news story broke so you can imagine that a user who viewed a page from one server, followed a link which landed on a different server, you'd like to have that link go somewhere useful even if a site update had occurred in the interim. In those days, we had to develop systems just to get new content onto a server. Whew!

After MSNBC, I returned to neurological testing at Cadwell. I'd previously written software for the first paperless EEG system, for measuring brainwaves. It was under DOS and we were still rolling our own network software so that you could record an EEG on one system and view it quickly on another. Once again graphics performance had been an issue and I wrote the actual waveform display code in low-level assembly language. The neurological test development gave me great insight into the functionality of neurons which I've incorporated into the current Brain Simulator software.

Today, I sit in front of a three-screen system with 64 cores and oodles of RAM. This CPU can execute up to 3 billion instructions in the 12 ms it took for one of those telephone relays to switch. Today's CPU is about ten million times faster than the 8080s we used for our first graphics terminals. These kinds of numbers are incomprehensible. If you could walk 10 millions times faster, you could walk around the world in about two seconds. If your brain were 10 million times faster, you could have a whole lifetime's experience in a few minutes. If you could do that, what would you do

with the rest of your lifetime? That's the type of question I'm working on today with the *Brain Simulator*. There's reason to think that the coming 50 years will bring the same factors of computer speed as the past 50 and the capabilities of such machines is beyond imagination.

Introduction

What makes the *Brain Simulator II* Unique?

Brain Simulator II is a free, open-source software project aimed at creating Artificial General Intelligence (AGI). Many important features set it apart from other Artificial Intelligence software:

1. *The AGI Strategy.* With the primary assumption that no one knows specifically how to create AGI, the *Brain Simulator* implements an experimental platform with a general AGI model which is easy to revise. Several spiking neuron models based on biological neurons combine with software "Modules" to create any desired functionality for rapid experimentation.
2. *The User Interface.* The graphical display of neurons and Modules lets users explore and modify the internal workings of the Network in real time.
3. *The Powerful Spiking Neuron Engine.* Tested with a billion neurons, the Brain Simulator can process up to 2.5 billion synapses per second on a desktop computer. Networks can also be distributed across a LAN with estimates of neocortex equivalence with only 160 servers.
4. *Software Modules and Applications.* To speed AGI development, over 50 Modules perform a variety of AI tasks. Combined with the Neuron Engine, applications already demonstrate vision, mobility, internal modeling, language, and planning.

The *Brain Simulator* implements an artificial entity named "Sallie" who lives in a simulated world and can integrate input from multiple senses. She can recognize objects with binocular vision and associate them with words she hears, plan a sequence of actions, and manipulate objects to achieve a goal. As she advances to understanding her world, interfaces already exist for cameras, microphones, and robotic control to bring AGI to life.

About the Brain Simulator II Project

This book is about *The Brain Simulator II*, a free, open-source software project aimed at creating an end-to-end Artificial General Intelligence (AGI) system. AGI is a loosely defined concept meaning

computer systems can respond in the same ways that as intelligent humans across a broad spectrum of situations. This contrasts with Artificial Intelligence (AI), which can often exceed human abilities but only in very limited situations (also called "narrow" AI).

In contrast with AGI, narrow AI performs poorly in the basic abilities common to any three-year-old. Just playing with blocks implies an understanding that physical objects exist and persist even when you can't see them, an understanding of gravity and the fundamental physics of solid objects, and a basic understanding of cause and effect and the passage of time—all absent from the typical AI.

The Brain Simulator II and its approach to AGI are significantly different from typical AI approaches. It is based on the reasoning that since we don't know precisely how AGI will work, and since our only AGI model today is the human brain, that studying brain functions and building biologically plausible approaches will lead to a quicker development of AGI.

My recent book, *Will Computers Revolt?,* discusses the potential *dangers* of AGI and how we can mitigate them. The fact that such a discussion is absent from this book is in no way an indication that AGI won't be dangerous or that caution is unnecessary. If *AGI danger* is your primary concern, I suggest you read *that* book instead. But *this* book *does* contribute to that conversation by showing methods, available today, which will contribute to AGI—meaning that AGI is not some far-off fantasy but will be upon us sooner than most people think. Further, the basic structure of AGI, introduced here, illustrates where AGI can be controlled and limited for the benefit of mankind rather than our demise.

Who Should Read this Book?

You will find this *Brain Simulator II* book interesting if you are:

- Interested in the capabilities and limitations of neurons
- Interested in Artificial General Intelligence
- A neuroscientist
- A cognitive scientist
- Interested in learning about spiking AI software
- A programmer interested in Artificial General Intelligence (AGI).

In addition to the explanations in this book, I have made numerous videos (available on YouTube) that address individual features of the project. I have included links at the end of each chapter to the relevant videos.

You will *not* find *The Brain Simulator II* interesting if you are looking for explanations of traditional Artificial Intelligence methods. The reason? The point of the *Brain Simulator* is to experiment with different ways of approaching Artificial *General* Intelligence. While there are numerous excellent books and software which explain and support Deep Learning and related AI approaches, in the intervening 40+ years since their introduction, traditional AI methods have produced interesting results but have not made much progress toward the common-sense intelligence enjoyed by any three-year-old.

The Structure of this Book

The book starts off with the development philosophy and the next six chapters (2-7) describe the *Brain Simulator* in greater detail, beginning with Neuron Models in Chapter 2. The *Brain Simulator* implements an array of millions or billions of simple neurons which can be computed in real time on a powerful desktop computer. Each neuron's behavior is dictated by the selection of one of several neuron simulation models. This chapter describes each of the models currently implemented and describes why you might want to use it. The internal code of a neuron model is not particularly complex and additional models can be added to the simulator as needed.

The distinction between biological neurons and traditional AI is highlighted in Chapter 3, "AI is Like Your Brain: DEBUNKED." This chapter shows how traditional AI techniques cannot possibly represent biological neurons because the underlying ideas are not possible in a biologically plausible world. Further, the simulator is designed around some features which are unique to biological neurons. While it would be possible to implement the idealized neurons of traditional Artificial Neural Networks (ANNs), such an implementation has not been pursued here because it would not be particularly efficient or enlightening.

For most people, it is not intuitively obvious how simple neurons can be harnessed together to perform useful functions. Chapter 4, "Applications of Neurons," shows several relatively simple combinations of neurons that work together to perform digital logic functions and several types of memory.

Arrays of neurons with their connections are stored in files called "Networks" which are described in Chapter 5. Within the *Brain Simulator*, information from one network can be included in other networks. This means that if you create some useful functionality in a small number of neurons, replicating this functionality many times is a fairly simple prospect. Most of the networks included with *The Brain Simulator II* perform relatively simple functions; the idea being that general intelligence will be created from millions of instances of a small number of unique, but fairly simple, neural circuits. This chapter includes a list of the Networks which are included with the program at the time of writing.

"Modules" form a key component of *The Brain Simulator II*, as explained in Chapter 6. It doesn't take much experimentation to learn that many functions which we presume to be simple are, in fact, difficult to implement in biological neurons. Enter the Module, which allows programmers to create custom computer code for any cluster of neurons.

This allows for three valuable uses: 1) Modules can implement the "rules" which govern the creation of connections for vast arrays of neurons; 2) Modules can be vastly more efficient than neurons for certain processes; and 3) Modules can be used to implement functionality for which neural implementations are yet to be determined. As an example: we don't know how binocular depth perception works in the brain, but within a module, we can use trigonometry to perform similar functionality. The chapter includes a list of the Modules which are included with the system at the time of writing.

Unique to *The Brain Simulator II* is the user interface as described in Chapter 7. You can examine and modify a neuron network while it is operating to design useful functionality. You can add or modify synapses and neuron parameters on-the-fly. And once created, neural circuits can be repeated, moved, saved, edited…anything you need.

Chapter 8 gives an overview of the programming interface. It explains how you might approach writing your own neuron models or Modules but details are left to the source code. Also, as the Brain Simulator is an open-source project, programmers are encouraged to add capabilities to the underlying user interface and Neuron Engine.

Chapter 9 explains some of the basic neuron networks included with the download. Highlights include several ways neurons can be configured as digital circuits, different ways that neurons can store information, and some of the capabilities and limitations of the various neuron models. Chapter 10 expands on this idea showing how variable synapse weights can expand the scope of neural circuitry.

The Universal Knowledge Store (UKS) is a set of Modules which can create relationships between disparate types of information. Chapter 11 explains how classic graph structures can be implemented in neurons (the genesis of the Modules). Chapter 12 then demonstrates how the UKS can be used with sample applications: 1) learning words associated with visual input and 2) navigating mazes.

One presumption of the *Brain Simulator* is that large numbers of neurons will be needed. Accordingly, the neuron models of the simulator have been optimized to work in multi-core and networked multi-computer implementations. Chapter 13 details the performance of the *Brain Simulator* in various configurations and projects the number of servers needed to emulate the entire human neocortex. Finally, Chapter 14 describes the current state of development and directions for future development.

So, the Brain Simulator is offered as an alternative. Accepting that no one knows precisely how to create a general intelligence system, the *Brain Simulator* is an experimental platform that begins with biological plausibility but enables shortcuts and extensions which can take advantage of the capabilities of today's CPUs.

Getting the Brain Simulator

You can download an executable free at http://brainsim.org. The download includes network files that demonstrate the features

described later in this book. You can do a lot with the *Brain Simulator* without any programming experience. The *Brain Simulator* runs on Windows 10, although some work has begun to create a Linux version.

If you are a programmer, you can also download the source code to glean more about how neurons and the simulator work. It is available at: https://github.com/FutureAIGuru/BrainSimII. Working with the source code to create your own modules or extend the neuron models requires Visual Studio; you can download the Community Edition free from Microsoft at: https://visualstudio.microsoft.com/downloads/.

If you are knowledgeable about the workings of today's Artificial Intelligence, you may be frustrated to discover that few of today's AI algorithms and constructs are included in the *Brain Simulator*. This is not because of some oversight but because today's AI has very little to do with the way a biological brain works. The algorithms of today's Artificial Neural Networks are based on a continuous neuron model which has little to do with actual spiking neurons. The primary backpropagation algorithm has no biological analog whatsoever. Both backpropagation and continuous neuron models *could* be implemented within the framework of the *Brain Simulator*'s neuron model and module structure but these have been extensively explored on other platforms.

The traditional AI approaches of today were developed into ANNs decades ago. If those algorithms had led to general intelligence in the intervening forty years, we would have discovered a different form of intelligence from the brain's. Having not achieved that goal, the *Brain Simulator* is an effort to return to the biologically inspired roots of AI.

Video Links

Many chapters are followed by links to related videos I have created. For a listing of all the related videos, go to: http://futureai.guru/videos

"Brain Simulator II Overview"

http://futureai.guru/videos?id=141

Chapter 1:
Brain Simulator II Strategy
OR
How to Create AGI

The Brain Simulator II contains innovative ideas toward creating Artificial General Intelligence (AGI). This is an ongoing research project. So rather than waiting for a "finished" version, I present the current state of development. I welcome suggestions, comments, and participation.

The Brain Simulator II is a PROJECT, not a PRODUCT. The project is FREE and OPEN SOURCE. Overall, the project is implementing the software architecture described in the book, *Will Computers Revolt?* and briefly described in this chapter.

This description is accurate as of the v1.0 release of *The Brain Simulator II*. The user interface and the Neuron Engine are relatively stable while the library of networks and software modules will likely change substantially and expand on a continuing basis.

Development Philosophy

We don't presently know precisely how to create Artificial General Intelligence. I'll describe many ideas in this chapter but their implementation is not well defined. Since we don't have all the answers, the *Brain Simulator* is an experimental platform. It's easy to try out new ideas and learn where improvements are needed. *The Brain Simulator II* allows experimentation with different neuron algorithms, different network designs, and higher-level Module software.

The source code download includes many features which are previews of future abilities, experiments that have been superseded,

and ideas that are still under development. The development technique follows the AGILE software development process of creating solutions for the simplest cases first. After these work, algorithms are generalized to handle more cases.

Productized software is certainly a possibility. But with an overall objective of creating AGI, there is no reason to expect that the project will be "complete" in the foreseeable future. Accordingly, it is written in such a way that there is no limit to the features which can be easily added.

The project adopts the incremental development aspect of AGILE software engineering. That is, it is important to develop software for a single use-case before addressing the myriad of use cases that an AGI is destined to encounter. This means there is already a single end-to-end path that *could* represent the mechanism of an AGI but with just a few use-cases.

The Reasoning Behind the *Brain Simulator*

You have to start somewhere! If we wait for a complete understanding, with robust mathematical models of AGI, we may never start. The *Brain Simulator* is based on reasoning about how general intelligence must work along with a recognition of the limitations of our knowledge.

1. No one knows, precisely, how to create AGI, hence a more experimental and iterative development approach.
2. Human intelligence and thinking occur in the brain and more specifically, the neocortex. This sets some limits on the size and complexity of the AGI problem.
3. Intelligence occurs in neurons as a result of their digital (spiking) function. This directs the *Brain Simulator* into areas of development outside of AI's classic perceptron/-backpropagation approach.
4. Intelligence has evolved since early man but the brain's structure has not. Rather than beginning with chess-playing, mathematics, or immense language skills, this project starts with basic techniques of finding one's way or understanding cause and effect.
5. AGI is not as big a development task as most think.

   a. DNA defines initial brain structure but not much DNA data is devoted to neocortex formation. Therefore, the brain (and an AGI) must be possible with repeating patterns of simpler neural circuits or simple rules which govern synapse creation.
   b. Brain capacity is bounded by neuron counts.
   c. Counts of sensory and motor nerves bound the incoming and outgoing data rates.
   d. These limitations set a maximum complexity for the brain and hence for an AGI.
6. AGI Requires Robotics. Without interaction with the real world, artificial intelligence will always be narrow. The real world is so variable and complex that simulators can speed development.
7. AGI can be created from existing hardware.
   a. Enough performance is available from today's hardware.
   b. Some subset of human performance could qualify as AGI.
8. AGI will not be like human intelligence. Human intelligence develops within the context of human goals, emotions, and instincts, which would form a poor basis for AGI.

Some of the reasoning above is currently subject to dispute and may eventually prove to be in error. But that's the point. The development of the *Brain Simulator* can help settle philosophical disputes one way or another. At the same time, the structure of the *Brain Simulator* is flexible enough to adapt to new information as it becomes available.

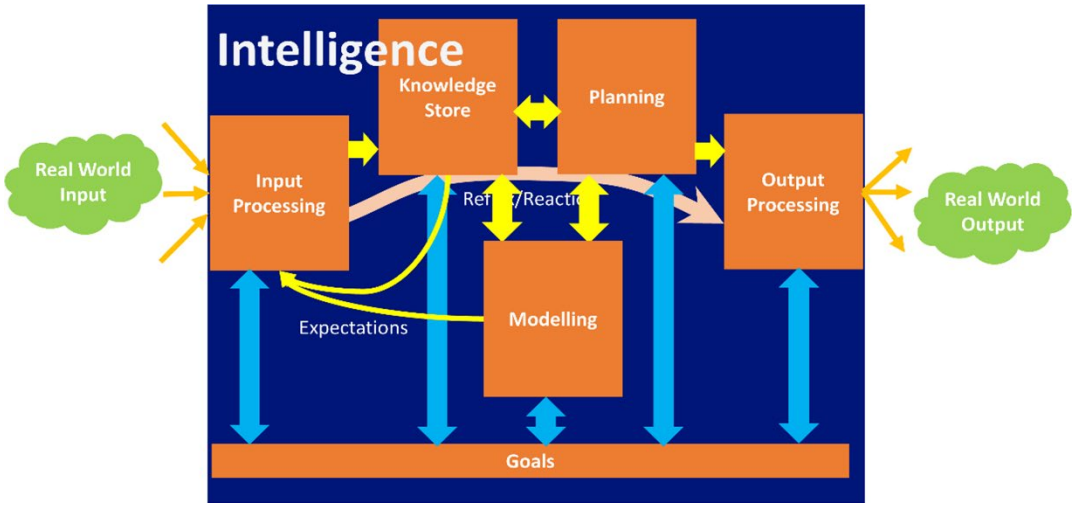
The Intelligence Model

The Brain Simulator II implements a simulated entity named "Sallie." Although it's fun to refer to any artificial entity by name and ascribe various intelligent attributes, Sallie has nowhere near the scope of capabilities needed for AGI. The following capabilities are *necessary* for AGI and may prove to be *sufficient* as well. Sallie can do all of the following things but in a limited way:

1. Sense her environment (input).
2. Act on her environment (output).
3. Have internal rules or goals.
4. Analyze inputs to make sense of her environment.

5. Remember (learn) combinations of inputs and actions and their qualitative results.
6. Internally model her environment in three dimensions.
7. Simulate possible actions and select for positive predicted results.
8. Perform these actions with sufficient speed and magnitude to respond to real-world conditions in useful timeframes.

Each of these has been implemented to some extent. For example, portions of the project can sense inputs from cameras and microphones and control a minimal robot, so Sallie can accept voice commands and "see" in some minimal sense. To date, however, *Brain Simulator* development has been primarily based on a simulated world…the real world just has too many variables and is not repeatable. The initial implementation uses a two-dimensional environment, although a simulated three-dimensional environment has been prototyped.



This block diagram, from Will Computers Revolt? *shows the major building blocks for a generally intelligent entity to implement the eight necessary components of intelligent behavior. It includes an Object Store (implemented as the Universal Knowledge Store explained in Chapter 10), input and output processing, and a 3D model of the entity's surroundings.*

To further illustrate the simplifications, consider that Sallie can only recognize two types of physical object. She can remember these in her internal memory so she knows where objects are, even when she can't see them. She can remember landmarks and use that

memory to plan her route to goals within a maze. She can act on her simulated environment by moving objects and, in one demonstration, can move an object to achieve a goal. She can learn to associate words with the objects she knows.

This is what is meant by an end-to-end prototype with just a few use-cases. The system can perform all the functions of AGI but only on a tiny number of data elements. Consider that before you learned to read, this page would have appeared as a mishmash of symbols—like looking at a page of Chinese characters (if you don't read Chinese). Sallie's perception is like that. She can see everything but only a few things make sense.

Many of the functions are hard-coded. For example, Sallie can learn to navigate a maze but she cannot learn about mazes because the maze-learning process is coded directly. The key is that the process of creating the software leads to learning about AGI. The maze software relies on the internal mental model and the storage of landmarks. Each landmark is a situation in the world at which Sallie must make a decision. She can recall the situation, the action she took, and the outcome. This concept of triples can be generalized to form the basis of planning and reinforcement learning for a much more complex AGI.

The Neuron Engine, User Interface, and Modules

To establish a bit of terminology, the *Brain Simulator* supports a vast array of simulated neurons connected by synapses. The function of simulating neurons and synapses is handled by the "Neuron Engine." The *Brain Simulator* user interface displays the neuron array (or some portion of it) so you can see what's going on and view or edit the pattern of connections. If the Neuron Engine is implemented on a different computer from the User interface, it uses an implementation called the "Neuron Server"—it's the same Neuron Engine, but with the added ability to handle connections and interfaces that cross machine boundaries on a network.

"Modules" are software shortcuts. A cluster of neurons can be considered a Module and neurons are then also under the control of the Module's software. Modules can be useful for input or output. For example, one module takes input from a video camera and fires its neurons as appropriate to represent the image. Other modules

are useful for computation. For depth perception, for example, we know your brain can merge signals from your two eyes and estimate the distance to an object. Rather than using neurons to perform this action, the Module uses trigonometry and all the power of the computer to perform a similar task more easily.

What, no Backpropagation?

The *Brain Simulator* makes no effort to compete with existing software packages which implement classic ANN algorithms. The intent is to break *new* ground with *new* algorithms. The thrust is to stay closer to biological plausibility since the human brain is the only working AGI model we have, at present.

After the following chapter describes the neuron models used in the *Brain Simulator*, I will return to this topic with a comparison of basic Neural Network algorithms vs. how your brain works in Chapter 3, AI is like your brain: DEBUNKED.

Video Links

"How to Create AGI"
http://futureai.guru/videos?id=127

"Brain Simulator II Presentation at AGI-20"
http://futureai.guru/videos?id=128

"Brain Simulator II Overview"
http://futureai.guru/videos?id=141

"How your Brain Works …in 5 Minutes"
http://futureai.guru/videos?id=109

Chapter 2: Modeling Neurons and Synapses

The *Brain Simulator* operates on an array of simulated neurons connected by simulated synapses. This chapter focuses on the function of neurons and synapses while Chapter 4 describes useful patterns of synapses which we'll call a "Network."

You'll get the most out of the *Brain Simulator* with a little background about how biological neurons function. This chapter is just an overview because biological neurons are immensely complex and variable. So I'll focus on a few principles and start with the simplest models.

The problem of understanding neurons is the same as with many biological systems—the chemistry which makes them develop and function is complex and may be more complex than is necessary to replicate with artificial means. To build an "artificial brain", we need to select which capabilities of neurons are necessary to the thinking process and which might be extraneous…necessary for the biological brain but not essential for thinking.

Consider creating an artificial heart…it's a pump. The individual cells of the biological heart work together in a coordinated way to pump blood. At the same time, these cells "know" how to grow within an embryo to create a heart and how to replace themselves if needed. Cells are also little metabolic machines that can convert the simple sugars and oxygen within the blood into the energy for contractions we know as a heartbeat. When we create an artificial heart, we don't worry about individual components of the pump carrying the instructions for how to build the entire pump or how to replace components when necessary. In the heart, it's reasonably clear that the pumping action can be separated from metabolism, "manufacturing", and "maintenance".

Neurons aren't so clear-cut. Neurons have a basic spiking action (which I'll detail momentarily), but like the heart, they also carry the mechanisms for developing and maintaining the brain. In the brain, though, the structure of individual neurons may also contribute to the thinking process. For example, which neurons contact one another is clearly essential to thinking. On the other hand, there are numerous types of neurotransmitters and synapses. Are these necessary to the thinking process or only for brain development? In addition to neurons, the brain contains glial cells which might contribute to thinking or might only be there to guide and maintain the brain's structure. It's hard to tell.

Neuroscientists have made great strides in modeling the complexity of neural function and several detailed neural modeling packages are available which simulate neurons near the molecular level. There are two problems inherent in this approach: First, a detailed computer model of a neuron contains numerous parameters and is necessarily slower, limiting the number of neurons which can be simulated at any given time. Second, with a focus on the detail of individual neurons, the overall functionality of the brain might be ignored. Similarly, a focus on an individual heart muscle cell sheds very little light on the overall function of the heart.

The *Brain Simulator* takes more of a "top-down" approach. Let's start with the simplest possible neuron models and see what can be produced with them. It's this approach that leads to the ability to simulate a billion neurons on a desktop computer and demonstrate how even larger networks can be simulated across multiple servers on a LAN.

When considering whether to add a capability to the neuron model, I evaluate how much computational power will be added to the simulation. For example, recent neuroscience experiments have shown that portions of a neuron can perform some basic computations. But those computations are the equivalent of adding intermediate neurons to the network—so it doesn't add anything to the overall capabilities of the simulator.

The Biological Neuron

To begin with the structure of the biological neuron, we'll start with the cell body, which is the center of the action. It has numerous

appendages, the "Dendrites," which can accept incoming signals from other neurons. It has a single long axon which branches at its destination to numerous synapses that connect to other neurons. The axon may be as long as needed to reach the desired destination and, as nerve cells within your body are a form of neuron, they may be over a meter in length. Within the brain, the average axon length is about 10 mm.

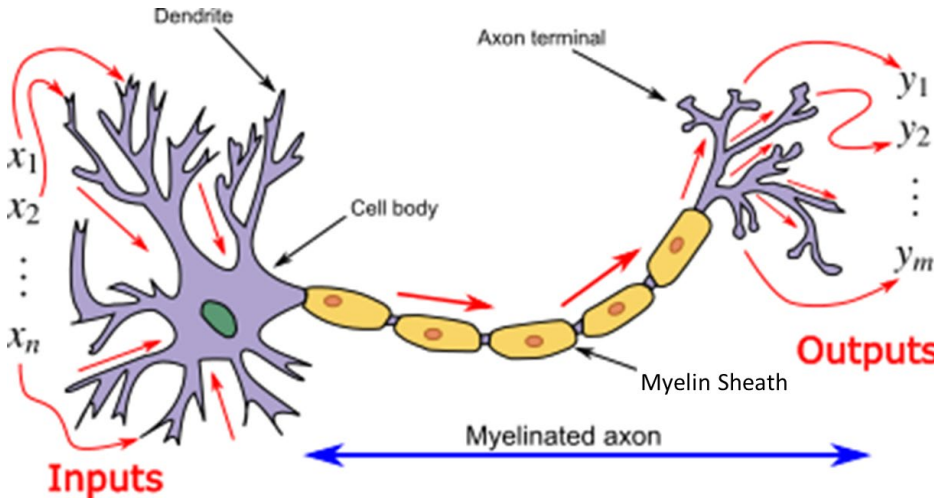
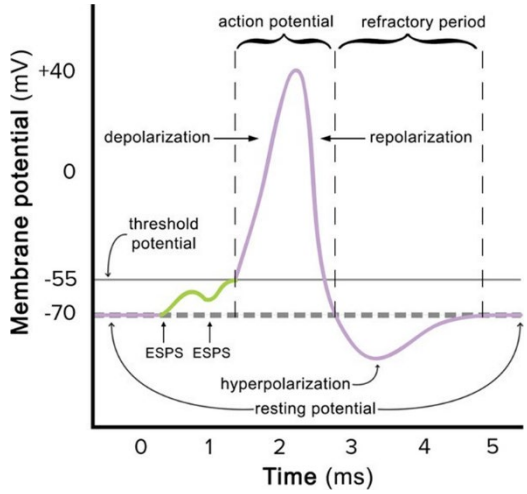


Diagram of a neuron showing "Inputs" and "Outputs," which are synapses and may count many thousands. The myelin sheath on the axon is only present on long axons, which may be 100mm long in the neocortex so this drawing is not to scale by several orders of magnitude. Shorter axons in the brain are not "myelinated," which makes them slower but more densely packed and still often hundreds of times longer than the cell body diameter. Diagram by Egm4313.s12 at English Wikipedia / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0)

The primary observable function of the neuron is to emit a spike which can be measured as a voltage pulse. This occurs when the internal charge exceeds a threshold—and that internal charge is accumulated from incoming neurotransmitter ions from adjoining neurons. The neuron has a measurable resting voltage. Incoming signals from synapses affect the voltage, with excitatory synapses increasing the voltage and inhibitory synapses decreasing it. When the voltage exceeds the threshold, a spike is emitted. It travels down the axon to the destination synapses, which contribute neurotransmitter ions to their target neurons. After the spike is

emitted, there is a "refractory period" during which all the neurotransmitters return to their original state so the process can repeat. During the refractory period, incoming signals are essentially ignored.

From the diagram, we can observe that the time from the onset of the spike to the end of the refractory period is about 4 ms. This means that the maximum firing rate of a neuron is about once every 4 ms or 250 Hz.



This diagram of an idealized neural spike shows how the incoming signals (ESPS—Excitatory postsynaptic potential) contribute to the membrane potential. When the membrane potential reaches the threshold, the neuron emits a spike and eventually returns to its resting potential for the process to repeat. https://www.researchgate.net/profile/Juan_Pedro_Dominguez-Morales/publication/329885401/figure/fig2/AS:707709062090765@1545742397755/Diagram-of-a-spike-generated-by-a-neuron_W640.jpg

Relative to a computer, the neuron is slow! Slow! SLOW! During the 4 ms of a neuron's firing cycle, my 4 GHz computer can execute 16 million cycles on each of its 64 cores. Add to that, neural spikes travel along the axon at a leisurely 2 mm/ms (walking speed). So a signal on an average 10mm axon arrives at its destination 5 ms after the spike is initiated at the cell body. The peak of the action potential is about 1 ms after threshold detection, so the fastest-possible signals through adjacent neurons (assuming no time for the axon delay) will induce a 1 ms propagation delay per neuron.

The Integrate and Fire Model

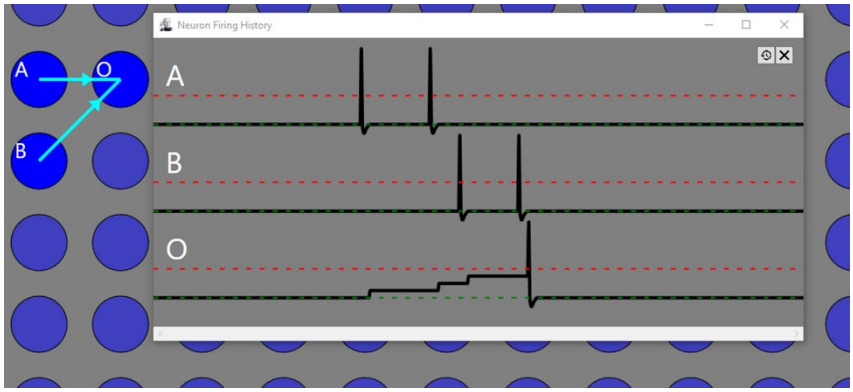
The *Brain Simulator* simulates a vast array of neurons…how vast depends on the machine and network limitations but it has been tested with a billion neurons. Each neuron can have any number of synapses connecting it to other neurons. Each neuron can be referenced by its "index" (or "Id") which is simply its position within the array. The array is presented in the user interface as being rectangular, implying row and column position relationships, but this is a convenience for visualization. For performance reasons, within the simulator, neurons are stored in a single, one-dimensional array and so have a single numeric Id.

Each simulated neuron has one principal function, it spikes, and two key features: 1) It carries an internal value and 2) a list of synapses to which it is connected. Each synapse consists of the Id of a target neuron and a weight. Both the neuron and the synapse carry additional, lesser features that will be covered later.

The primary function of a neuron is to emit a spike dependent on the inputs it receives. The way it "decides" to emit a spike depends on the "model." The *Brain Simulator* supports any number of models and new ones can be added easily. We'll start with the simplest, Integrate and Fire ("IF").

The Neuron Engine can evaluate every neuron just once in a "cycle". For now, we'll assume that the cycle time is equal to the refractory period (4 ms) and that all neuron firing is synchronized. Later, I'll explain how to relax this limitation for greater precision.

Considering the basic IF model, here is how the neurons work. Each neuron is evaluated to determine if the internal value exceeds a threshold. If it does, then the synapses list is processed and the weight of each synapse is added to the internal value of the corresponding target neuron. The synaptic weight can be positive or negative, either contributing to ("excitatory") or subtracting from ("inhibitory") the neuron's internal charge.



Illustrating the "Integrate and Fire" model. The randomly firing neurons "A" and "B" are connected to neuron "O" with synapse weights of 0.25. In the diagram above, whenever A or B spikes, you can see the internal charge of O increase somewhat (the bottom line in the History window). On the fourth incoming pulse, O reaches its threshold, emits a spike of its own, and resets its internal charge to 0.0.

For convenience, the resting state of the neuron is defined as 0.0 and the threshold is defined as 1.0—these are different from the actual voltages observed in biological neurons and are arbitrary values since all other values within the system are scaled accordingly. The internal value is represented by a floating-point number. This is somewhat counter to the biological observation that the internal value cannot represent very many different values (covered in the "Differences" subsection later).

It is easy to see that a synapse with a weight of 1.0 is sufficient to cause the target neuron to spike (in the absence of any other input). To illustrate that the threshold is arbitrary, if the threshold were defined as 2.0, then a synapse of weight 2.0 would be required to cause a spike but everything else would work in exactly the same way. Synapse weights are, likewise, floating-point numbers. Again, experimentation with biological synapses shows they are limited to as few as 100 discrete values. Because today's computer has been optimized for floating-point calculation, the performance cost of using floating-point numbers is nominal. On the other hand, the memory cost is significant. A single byte would be all that's required to represent a realistic charge value or synapse weight, while the floating-point number requires four times the memory.

A synaptic weight of less than zero is inhibitory. Because synapse weights in the *Brain Simulator* are floating-point numbers, the weight can glide freely from excitatory to inhibitory and back again. In biology, this is more difficult because such a synaptic sign change requires the use of different neurotransmitters and neuroreceptors involving different ionic charges.

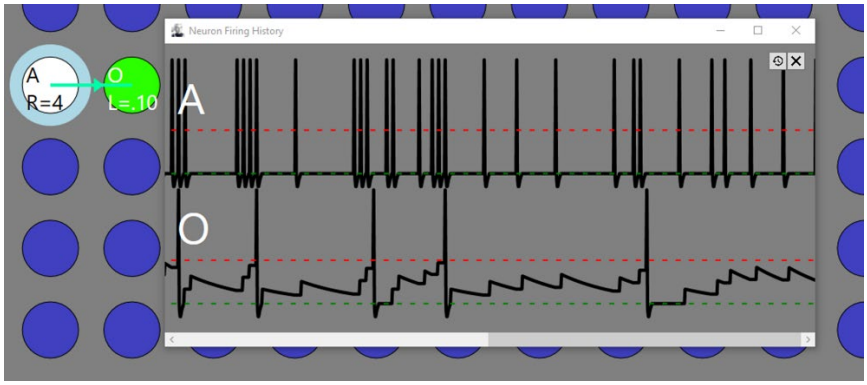
Considering just a single input, you can see that for an incoming synapse weight of 1, the output spiking frequency will match the input. Every time the stimulating neuron spikes, the synapse contributes enough charge to cause an outgoing spike.

If the weight is between 0.5 and 1, then it will take two incoming spikes to create an output spike. For example, with a weight of 0.8, the first incoming spike will increase the internal charge to 0.8 and the second will cause the internal charge to increase to 1.6 which exceeds the 1.0 threshold, will cause a spike, and will reset the internal charge to 0. With a weight between .33 and .5 it will take three incoming spikes to cause an output spike…and so on.

The neuron is acting as a frequency divider. This illustrates an important limitation of neurons (both biological and simulated). Since there are no partial spikes, a neuron always requires an integral number of spikes to reach its threshold (or exceed it). This makes it impossible to process a repetitive signal from a single neuron by considering its frequency as a floating-point number. The most precise frequency processing is to divide the incoming frequency by two—and not many stages of such division will result in a signal which is too slow to be useful for thinking. There are ways to circumvent this limitation with the use of multiple neurons as is demonstrated in the “Basic Neurons” network.

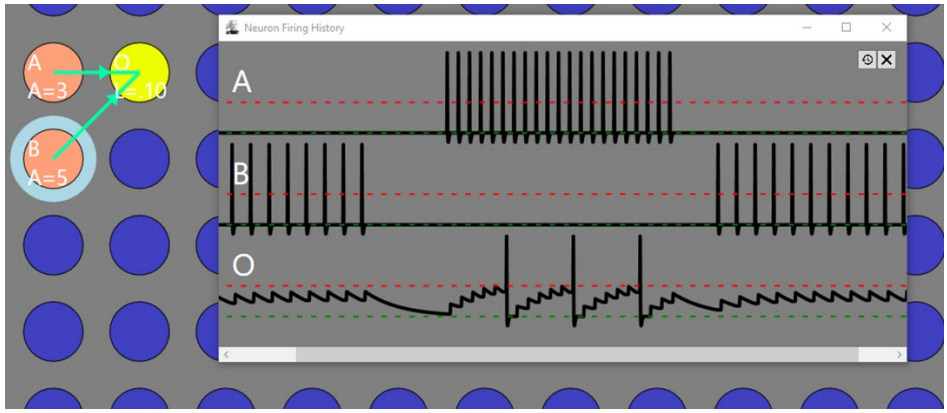
Adding Leakage

The idealized neuron of the IF model will store its internal charge value indefinitely. In the real world, charge leaks away at some rate, and in neurons, leakage can be significant. This leads to an extension to the model, Leaky Integrated and Fire (LIF).



With the LIF model, incoming spikes still contribute to the internal charge of a neuron but between incoming spikes, the charge "leaks" or decays away a little. If the incoming spikes stop, the internal charge will gradually return to 0.0.

In the model, the leakage rate defines the fraction of charge which is subtracted in each cycle. In the *Brain Simulator*, you can set the leakage rate for individual neurons. If you set the leakage rate to 0, then the LIF neuron acts as an IF neuron because there will be no leakage. If you set the leakage rate to 1, then no charge is maintained from one cycle to the next.



The leakage rate can be used as a "high-pass" filter in that if the incoming spike frequency is high enough, O will spike periodically (left) but if it is lowered, O will never spike (right).

In between, there is an interesting and useful circuit. You can see that if the rate of incoming spikes combined with their synapse weights exceeds the leak rate, the neuron charge will increase and the neuron will eventually spike. If the incoming spike rate is less than the leak rate, charge will be draining off as fast or faster than it

arrives and the neuron will never fire. This means that any neuron using this model can act as a frequency-threshold detector. It will only fire if the incoming frequency exceeds some given value (dependent on the leak rate). Further, with the added frequency-division described above, once the neuron begins spiking, its output frequency will be proportional to the input frequency.

If the incoming spikes are connected by a synapse of weight 1, the neuron will fire on every incoming spike, regardless of the leakage rate and so it is no different from an IF neuron. If the incoming spike is connected by a weight of 0.75, then the neuron will fire after two spikes but ONLY if the second spike arrives soon enough that leakage has not drawn the internal charge below 0.25. Above this rate, the neuron will fire on every other incoming spike.

Randomness and Noise

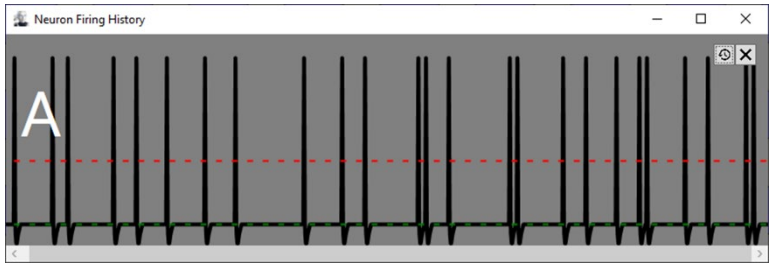
The models presented so far create ordered, predictable results. When we probe the brain and look at neural signals, there appears to be disorder and randomness. To consider a conceptually simple signal coming to the brain, think in terms of a signal coming in from a single retinal cell that spikes faster with brighter light. Ideally, the frequency of spiking would track nicely with brightness at a particular point. Observation, though, shows a lot of "jitter" and only the average spiking frequency over a longer period tracks well with the intended signal. Whether this variability is noise or is additional signal encoding is not known. There is an ongoing discussion about whether this randomness is extraneous or essential to the thinking process.

The spiking frequency of a biological neuron shows a considerable amount of variability. This is most likely due to electronic or chemical "noise" and doesn't represent the signal.

Random Neurons

As the random character of neurons is not precisely known, the *Brain Simulator* has made a first step by incorporating a "Random" neuron model. Neurons with this model act as IF neurons except that

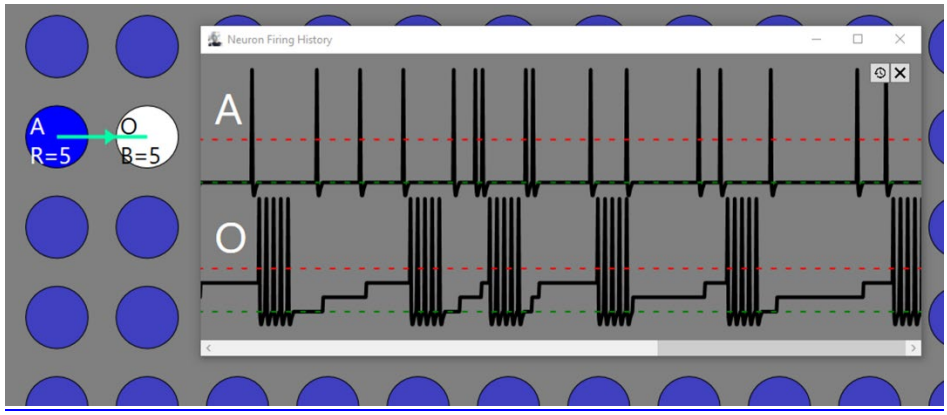
in the absence of any stimulation, the neuron will fire at a random interval with a given mean and standard deviation. The overall frequency of random firing is governed by the mean. If the standard deviation is set to 0, the neuron will fire at a constant rate given by the mean. If the standard deviation is negative, the neuron will be disabled. The Random neuron model is used internally to represent always-firing neurons. A random neuron with a long mean will act generally like an IF neuron except that it will occasionally emit a random spike.



The spiking pattern of a random neuron with a mean of 5 and a standard deviation of 2. Setting the standard deviation to 0 will cause the neuron to fire at a constant rate.

The Burst Neuron

Some biological neurons appear to fire bursts rather than individual spikes. Within the *Brain Simulator*, the "Burst" model performs a similar function. There are two parameters, one of which governs the number of spikes in the burst and the other the rate of spikes (in cycles). In other respects, the Burst neuron model acts as an IF neuron.



This timing diagram illustrates the burst neuron, O, driven by IF neuron A with a synapse weight of 0.5. It takes two incoming spikes from A to cause O to fire a burst. The number of spikes in the burst is set to 5 and the rate is set to 1 so the burst fires at the maximum neuron firing rate.

The Always Firing Neuron Model

For convenience, this neuron model fires at a consistent rate every *n* Neuron Engine cycles. The "Always" neuron model can be useful for some digital circuit prototypes where continuous firing is needed. In practice, it is the same as the Random model with a standard deviation of zero.

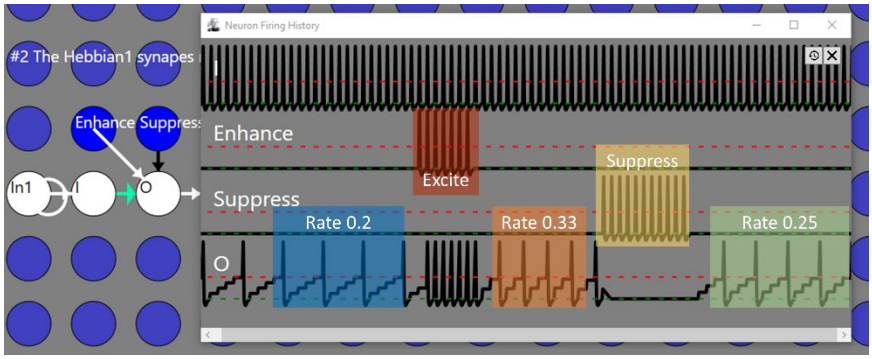
The Hebbian Synapse

Thus far, the models have relied on synapses that have fixed weights but there is ample evidence for synaptic plasticity. This opens the door to the memory mechanism where information is stored in the weights of synapses (described below).

This learning process was initially described by D. O. Hebb and is also called Spike-timing-dependent plasticity. In general, if one neuron has a synapse that targets another and it fires immediately prior to the target neuron firing, we could assume that the first neuron caused the firing of the second (or should have if it didn't). Therefore, the synapse weight should be increased; otherwise, the weight should be decreased.

This is more generally called spike-timing-dependent plasticity but I'll continue to call it "Hebbian". Based on the interspike timing,

Within the *Brain Simulator*, only certain synapses are designated as Hebbian. One can see that if the "control" synapses from A and B were also Hebbian, things would be more complex. So, the structure of a circuit has fixed-weight synapses and only "content" synapses are plastic. For testing purposes, one can select an area of a network and reset all the Hebbian synapses within that area without affecting the fixed-weight synapses. This can be convenient for resetting memory.



With a Hebbian synapse connecting "Clock" to "Out," spiking on B inhibits Out and causes the synapse to weaken. Spiking on A stimulates Out and causes the synapse to strengthen. The approximate weight of the Hebbian synapse can be inferred from the spiking rate of Out.

In the same way that the action of a neuron within the *Brain Simulator* is governed by the "Model", synapses, too, have a model which selects their action. One specific model, "Fixed," does not allow for any plasticity and this is the default model for synapses. Therefore, only certain synapses are designated as Hebbian. One can see that if the "control" synapses from A and B were also Hebbian, things would be more complex. So, the structure of a circuit typically has fixed-weight synapses and only "content" synapses are plastic. For testing purposes, one can select an area of a network and reset all the Hebbian synapses within that area without affecting the fixed-weight synapses. This can be convenient for resetting memory.

A precise formula for the amount of increase or decrease of biological synapse weights is not known and within the *Brain Simulator* is controlled by a lookup table which is subject to change. The selection of the model simply selects the lookup table to use. As of this writing, there are three weight control tables.

- Binary: simultaneous firing causes the synapse weight to be set to 1. Firing the target without the source sets the weight to 0.
- Hebbian1: Weights range from 0 to 1 such that all single synapses are stable (see below).
- Hebbian2: Weights range from -1 to 1. Weights are varied so pattern recognition is learned with an arbitrary number of input spikes.
- Additional models are anticipated.

The weights in the Hebbian1 model are set so that fractional-value weights of individual synapses will be relatively stable. That is, a Hebbian Synapse with a weight of 0.25, for example, will remain near its current weight. Since the 0.25 synapse will cause the target to spike on only every 4th cycle, the weight increase on firing must be four times the weight decrease when not firing for the synapse weight to remain stable over time. This leads, overall, to synapse weights which can increase faster than they can decrease. Note in the illustration that the synapse weight is reduced with 8 spikes but restored with only 5.

While the ratios of weight increases to decreases are dictated by the desire for stable synapse weights, the absolute values are dictated by the desire for weights to change as quickly as possible to facilitate rapid learning. As currently implemented, a 0.0-weight synapse can be brought to a weight of 1.0 with 11 spikes but returning it to zero requires 33. Using a 4 ms neuron cycle time, this means that it takes up to 44 ms to set a 0.0-weight synapse to an arbitrary value. While this timing is plausible, note that it restricts the synapse weight to only 11 discrete values. Increasing the number of possible synapse weight values would require smaller weight changes for each spike. This can be done by changing entries in the lookup table but would result in correspondingly slower learning rates.

Note also that even when attempting to set a synapse weight to some precise value, there is no practical way, within the network, to learn what the precise value is. A synapse weight of 0.5 will cause a target neuron to spike on every other cycle—but so will a weight of 0.75 or 0.9. So you may observe that a neuron is firing every other

cycle but this only gives a general indication of the aggregate incoming synapse weight, never precise values.

Synapse weights do offer higher precision the smaller they are, but, once again, this leads to a corresponding reduction in speed or increase in complexity as it takes progressively more incoming spikes to stimulate a neuron to fire.

The limitation on setting precise synapse weights and subsequently determining what they are will be revisited in Chapter 3 ("AI is Like Your Brain: DEBUNKED").

Adding Timing (Refractory & Propagation Delays)

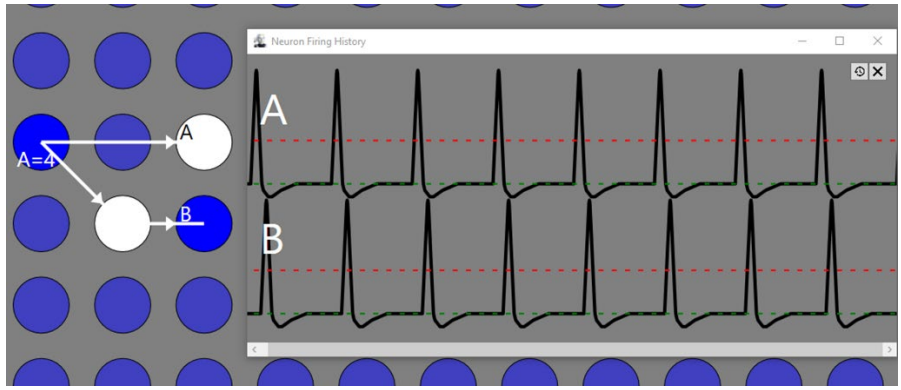
Thus far, the demonstrations have presumed that the Neuron Engine cycle time is equal to the neuron's refractory period. That is, that a neuron can fire once for every Neuron Engine cycle. This simplification works for a large number of circuits but in some areas of the brain, more timing is important.

Consider again that you have an incoming signal, such as from a retinal cell, which spikes at different rates proportional to the brightness at a particular point. The maximum firing rate of the neuron is 250 Hz so if we were to be able to register an image in a 50 ms timeframe (not an unreasonable assumption), there would be a maximum of 12 spikes in that timeframe. One could imagine that the one-pixel brightness signal varies with time and that the number of spikes over the past 50 ms represents the brightness at any given time. Also, it's not unreasonable that a brightness signal is limited to fewer than 12 different levels as we know that it's difficult to detect more than 12 levels of gray (see *Will Computers Revolt?* Chapter 9). I've already alluded to a neural circuit that balances neuron leakage rate with incoming spike rate to filter the incoming spike rate. But with a 4 ms Neuron Engine cycle time, it isn't possible to represent a neuron spiking every 5 ms or 6 ms, for example.

Refractory periods

It turns out that the only thing which relates the Neuron Engine cycle time to real-world neuron spike times is the refractory period. We've assumed that a neuron can spike in every cycle, therefore, the cycle time must be 4 ms. By setting a different refractory period, we can define the cycle time to be as precise as we like. If we set the refractory period to be 4 Neuron Engine cycles, then a neuron can

fire once every 4th cycle, so the engine cycle time must be 1 ms. If we say that the refractory period is 40, then the neuron can only spike every 40th cycle so the cycle time is 0.1 ms.



Neuron B spikes one Neuron Engine cycle after A. With a refractory period of 4 cycles, we can see that B fires during the refractory period of A…only 1 ms later.

Since the lion's share of computing power in the Neuron Engine is used only when neurons spike, changing the cycle time does not dramatically increase the amount of CPU time required. Increasing the refractory period can change the number of cycles between spikes but does not significantly alter the number of spikes that must be processed.

Note that the leakage rate of LIF neurons is in the amount of leakage per engine cycle so as the refractory period is reduced, all leakage rates must be correspondingly reduced as well.

Axon Delays

A target neuron spikes in the engine cycle immediately after its threshold is reached and this is not unreasonable for a 1 ms cycle time. For a 0.1 ms cycle time, however, things will be too fast to be biologically plausible. The key thing to understand is that, from a simulation perspective, the time that a spike is initiated at the cell body is not nearly as important as the time that the spike arrives at the target synapses.

Since neurons in the brain can connect to others a considerable distance away, and the neural spike travels along the axon so slowly, the pulse might arrive at the target synapses anywhere from 1 ms to tens of milliseconds later. This is ignored in the models presented so far but is significant for certain networks like determining the direction of incoming sound, and may be important in other future

development. To compensate, the LIF neuron model carries a parameter of "Axon Delay" which determines the number of cycles that it takes for a spike to reach the target synapses.

Whereas the refractory delay is the same for all neurons, the Axon Delay must be set specifically for individual neurons as they may have different physical axon lengths.

So if the refractory period is set to 40, implying that the engine cycle is 0.1 ms, it would be reasonable to set the Axon Delay to 20 (for example) so that neural spikes would arrive at target neurons 2 ms after the threshold is reached.

With appropriate settings of the Refractory Period, Axon Delay, and Leakage Rate, a reasonably accurate representation of precise neural timing can be achieved.

Short-Cut Models

There are inconveniences in using neural spikes to represent information and computers are adept at many things that are difficult to implement in spikes. Three examples are given here:

Color Neurons

The eye receives light and emits spikes down the optic nerve corresponding to (among other things) the color detected at any point in the visual field. The eye has separate sensors to detect different colors (red, green, blue, and gray intensity) and signals from these neurons seem to remain separate through the optic nerve. The computer stores the RGB (or aRGB) triples in single memory words and decodes them as needed—a process that is not biologically plausible. For convenience, there is a "Color" neuron model. It does not spike but simply stores an integer value that could represent a color. As an added convenience, the color of the neuron display in the user interface is governed by the RGB value, so if you have an array of Color neurons, you can see the color image in the user interface. Also, the internal value is displayed or modified in hexadecimal as the "Charge" of any Color neuron.

The content of a Color neuron is only useful to Modules because it generates no spikes. To extract a color from the Color neuron, you need a few lines of code which will mask off the portion of the color signal which is of interest. These components of color can then be

handled by more biologically plausible neurons. The conversion from a single Color neuron to the spiking rates of four IF neurons (representing red, green blue, and gray intensity) is demonstrated by the code in the ModuleColorComponent Module.

FloatValue Neurons

Like Color, there are times that a high-precision floating-point number is useful. Biological neurons have a limited range of possible, discernible values, because high noise levels (and leakage) limit the number of discrete values a neuron might represent. Also, representing a signal in a neuron's spiking rate is limited by the speed at which a signal must be represented. Like the Color neuron, neurons with the FloatValue model do not spike and must be accessed via a software module.

Neuron Labels

Every neuron may carry a text label, and these are typically used just to display in the user interface in order to keep track of which function is being handled where. In the demonstration networks, some neuron labels are used to indicate the function of a neuron or to refer to it while others are used simply as notation labels within the network.

There is no limitation, however, in the way a Module can manipulate the label so it *could* also be used to store a text string which can be used by other software in a Module. As an example, neurons in speech-recognition are given labels that correspond to the words or phonemes they represent. Also, a Module may reference any neuron in the network by its label as well as by its Id to, say, add a synapse or read or set a value.

Since neurons are sometimes referenced by Id and may also be referenced by label, numeric labels (which might be ambiguous) are automatically prepended with an underscore ("_") which does not show in the neuron display. Labels don't need to be unique but this can cause issues when subsequently referencing neurons by label so there is a warning when setting a neuron label if it is already in use.

Differences between Brain Simulator and biological neurons

For the most part, anything which can be done in *Brain Simulator* neurons is plausible in biological neurons. As mentioned previously, biological neurons represent a complex soup of chemicals and it is

difficult to distinguish between the components of neural activity that are essential to intelligence and those which are artifacts of their biological nature.

This section describes some of the many differences between the *Brain Simulator* and its biological counterpart.

Use of Floating-point Numbers

Within the *Brain Simulator*, neuron internal values and synapse weights are processed as floating-point numbers. Because of decades of CPU performance optimization, today's CPUs handle floating-point numbers nearly as fast as integers, and if one wished to be more biologically accurate and were to implement a system limiting things to 256 discrete values (for example), the memory requirement would go down by a factor of 4 but processing time might not improve.

The use of floating-point numbers allows for minute differences in various values to impact the result. For example, small differences in synapse weights can be used to encode information while this is not possible in biological neurons.

Noise

Brain Simulator neurons can be noise-free and synchronized. Unless noise is deliberately introduced with Random neurons, the operation of a network will be absolutely consistent and repeatable.

Reliability

For all practical purposes, neurons in the *Brain Simulator* are completely reliable whereas neurons in the brain are not. Most of the networks in the *Brain Simulator* depend on this reliability and the failure of any single neuron or synapse might cause the network to fail. To make a *Brain Simulator* Network more resilient to neuron failure would require significant design additional effort and many additional neurons and synapses.

High synapse weights and multi-synapse Equivalence

Most networks rely on synapse weights which are significant relative to the threshold, so a small number of incoming spikes can cause the target neuron to spike. While we can assume that individual biological synapses have a much smaller maximum weight, a single high-weight synapse is functionally equivalent to a number of smaller-weight synapses in parallel.

The *Brain Simulator* only allows a single synapse between any two neurons and we simply presume that this represents the aggregate weight of a larger number of parallel synapses. As a consequence, the number of synapses required to implement any specific function is much smaller than the number of synapses observed in biological neurons.

Creating New Synapses

In the brain, synapse weights can be changed in milliseconds. On the other hand, creating *new* synapses is very slow—observed over periods of hours or days. In the *Brain Simulator*, creating a new synapse is only slightly slower than changing the weight of an existing synapse. Although new synapses can only be added rapidly by Modules, this eliminates the need for huge numbers of synapses with near-zero weights which are in place in the brain as placeholders waiting for their weights to be increased so they can be significant.

Once again, the number of synapses required for a learning function in the *Brain Simulator* is much smaller than the number of synapses observed in biological neurons.

Sign Transitions of Synapses

In the *Brain Simulator*, since synapse weights are represented by floating-point numbers, the only difference between an excitatory synapse and an inhibitory synapse is the sign of the weight. In biology, different synapse types must use different neurotransmitters with different ionic charges. As such, the biological equivalent of the *Brain Simulator*'s smooth glide from excitation to inhibition must involve setting the biological excitatory synapse weight to 0, then increasing the inhibitory synapse weight. This means that in a general learning environment, there must be two biological synapses to be equivalent to one *Brain Simulator* synapse.

Once again, the number of synapses required for a learning function in the *Brain Simulator* is smaller than the number of synapses observed in biological neurons by a factor of two.

Synchronization

In the brain, all neurons can work asynchronously but in a computer, things are necessarily more sequential so synchronization is necessary. Imagine that two neurons fire at the

same time and both connect to the same target neuron, one with a weight of +1 and the other with a weight of -1. In your brain, the target will never fire. In a simulator, if the +1 synapse is processed first, the target neuron will fire but if the -1 is processed first, it will not.

To eliminate this problem, the Neuron Engine algorithm imposes a discrete time step and two-phase processing. Within a time step, every neuron has the chance to fire just once, then every firing neuron processes its target synapses. By setting a long refractory period (a short cycle time), the impact of synchronization can be minimized but most of the *Brain Simulator* networks are built with a Refractory Period of 0. This is sufficient to represent a huge number of network capabilities but it does impose some timing limitations.

Performance

A considerable amount of code is required to make the neuron array work properly in a parallel environment on a multi-core system or across multiple computers in a networked system. Additional code was needed to make the system FAST. The actual speed measurements are included in Chapter 11 on performance.

As a point of comparison, the 450-neuron BasicNeurons network, which is set with a refractory period of 0, can process engine cycles in 0.04 ms—100 times faster than biological time. In general, the processing requirement goes up with the number of neurons that are firing. As neocortex neurons fire, on average, once every six seconds, simulation of the entire neocortex on an array of today's high-performance servers should be possible.

The Array Structure

Within the *Brain Simulator*, each neuron has an ID which is its index in the neuron array so it is a simple matter to indicate any specific neuron. In the brain, no such addressing scheme exists. In the simulator, you can add a synapse between neuron ID=1234 and neuron ID=5678. But in the brain, you can only add a synapse based on some function of physical proximity or some function of current firing state.

This means that your brain might create hundreds or thousands of synapses and then winnow these down to those which turn out to be significant.

Video Links

"How Your Brain Works: Part 2 Neurons"
http://futureai.guru/videos?id=107

"Introducing the Brain Simulator II"
http://futureai.guru/videos?id=112

Chapter 3:
AI is NOT Like Your Brain

Sometimes at the beginning of a movie, you see something like: "Inspired by true events" or "Based on a true story". Saying that "Artificial intelligence is like your brain" is a lot like that. It starts with a few facts, then the rest of the movie goes off in a different direction.

Your brain is so different from AI's Artificial Neural Networks (ANNs) that in this chapter I'll focus on just three areas:

- ANN neurons aren't like biological neurons.
- Artificial synapses are only a little like biological synapses
- Backpropagation, the mainstay of AI learning, has no biological analog whatsoever.

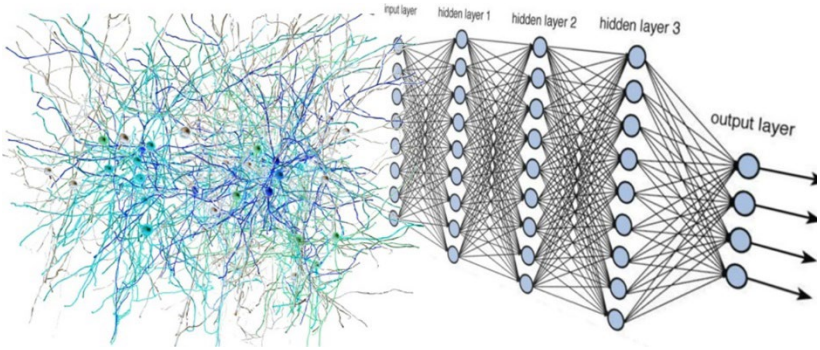
Of course, there is another branch of AI, Symbolic AI, but it makes no pretense of being like your brain so we'll bypass it for this conversation.

The reason this is an important topic is that in the search for Artificial *General* Intelligence, today's crop of AI applications show very little aptitude in areas where any three-year-old can excel, things like: common-sense understanding, cause and effect, the passage of time, or gravity or spatial relationships.

We have an excellent example of general intelligence in the human brain. So, Like the Wright Brothers who analyzed birds when designing the first airplane, let's compare the human brain's similarities and differences with today's ANNs.

Neural networks are *so* different from the way your brain works, let's start with the lone similarity…the general concept. Both have things called "neurons" interconnected by weighted "synapses" and the state of a neuron impacts the states of neurons to which it is connected.

But there, the similarity stops. Biological neurons don't appear in orderly layers with orderly connections between one layer and the next like in Neural Network diagrams. Instead, your brain has a tangle of interconnections that we have yet to unravel.



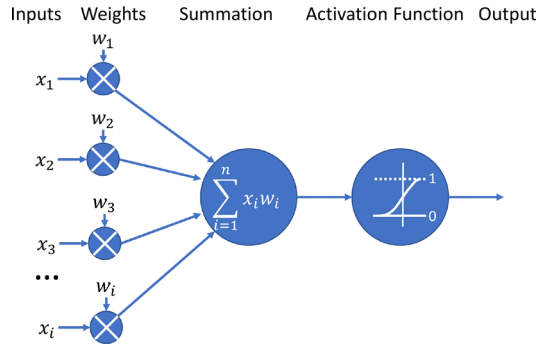
Your brain is a tangle of connections like the artistic rendering on the left while Artificial Neural Networks portray an orderly, layered connection structure.

Neurons

Computer models of biological neurons can be complex but fortunately, the simplest neuron model, Integrate and Fire, is sufficient to show the limited relationship with the ANN.

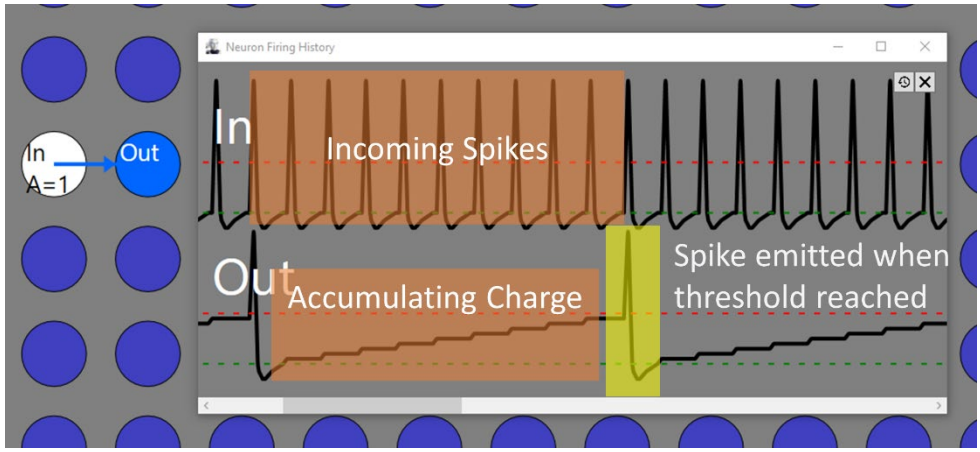
As described in the previous chapter, the biological neuron accumulates charge from incoming synapses and emits a spike when a threshold is reached.

Let's turn to Artificial Neural Networks. If you're at all familiar with them, you've seen the weighted-sum formula numerous times. It's a useful formula but it doesn't match the IF neuron model—and more sophisticated models diverge even further.



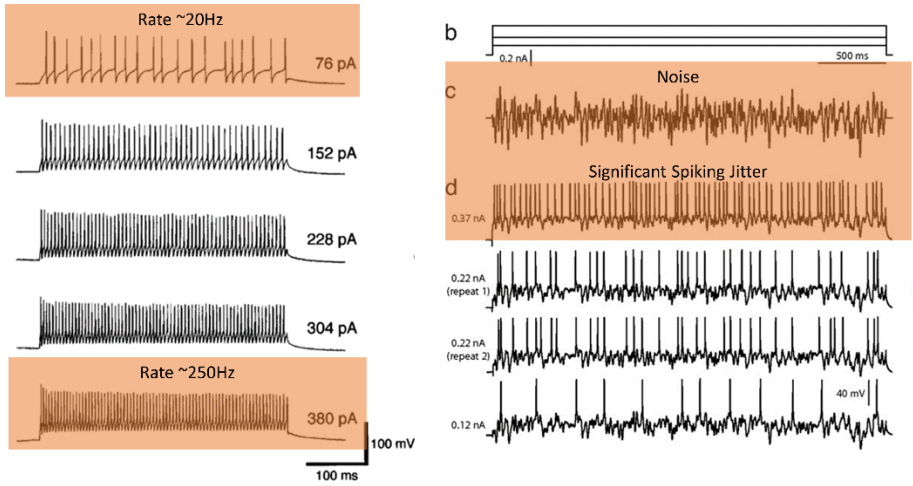
The classic Artificial Neural Network neuron algorithm takes all inputs, multiplies by weights, sums them, and then uses an activation function to create the output value.

Here's the initial problem. The biological neuron is a spiking device that cannot and does not output an analog value like the x's and the output value in the illustration. Instead, neuron values are binary—either there is a spike or there is not. In a few cases, AI experts counter by saying: "No problem, just set the activation to a step function so the output will be 1 or 0." But this ignores the accumulated charge from one cycle to the next, so we'd also need to add internal memory—not reflected in the neural network model. With enough correction, the formula can morph into the biologically plausible one used in the *Brain Simulator*—but by then, any relationship with Artificial Neural Networks is lost.



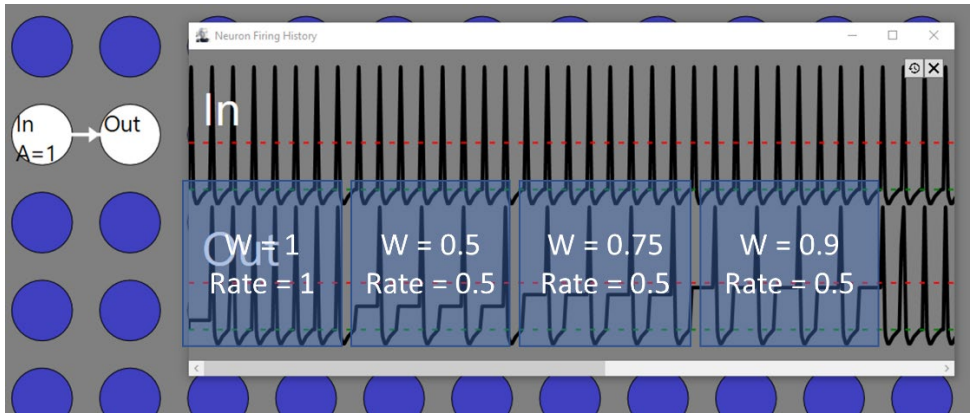
This is a timing diagram of IF-modeled neurons, the upper having a synapse connected to the lower. Each spike of the upper neuron contributes to the charge of the lower neuron until the threshold is reached. Then the lower neuron emits a spike of its own and the process can repeat.

OK, so many AI experts try to rectify things by saying that the x's don't represent individual spikes but represent the spiking *rate* of the neuron—the idea is that the *rate* could vary continuously. In practice, though, the spiking rate cannot vary continuously because the neurons have a maximum spiking rate of about 250 Hz and neural signals cannot be useful below about 20 Hz. In between, high noise levels in the brain limit the number of different rates that can be reliably represented. But I won't dwell on these practical problems and instead move on to an even more fundamental issue.



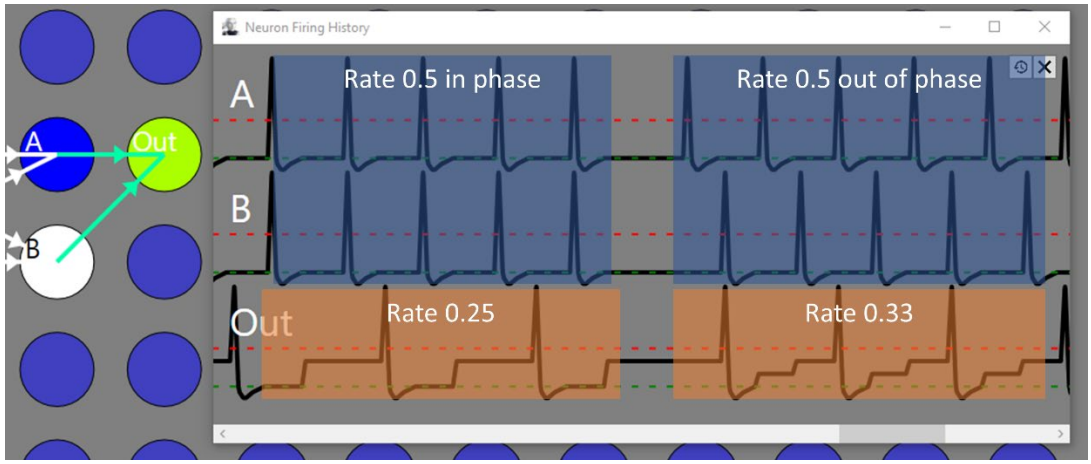
In the real world, biological neuron firing rates can't represent very many different values because they have minimum and maximum useful rates and the noise levels in the brain are high, limiting the number of different rates that can represent a value reliably.

When you consider how changing an incoming synapse weight affects the spiking rate, you see that the biological spiking neuron simply doesn't match the ANN formula. This is because these pesky biological neurons are SIMPLY NOT LINEAR DEVICES.



With an input spiking at a constant rate (1.0) while we vary the incoming synapse weight, you can see that the neuron's spiking rate doesn't match the ANN formula. With a weight of 0.9, the formula defines a rate of 0.9 while we observe a rate of 0.5.

For a somewhat more complex example, consider that biological neurons can be affected by the timing of the incoming signals received. This example is easily replicated within the *Brain Simulator* and shows that two signals, both at a rate of 0.5, can result in different spiking rates dependent on the phase of the two signals.



This Network illustrates another shortcoming of the ANN formula. Both the input signals, A and B, are firing at the same rate of 0.5. But the output rate can be 0.25 or 0.33 depending on whether A and B are firing at the same time or alternately.

When we look at the formula, signal phase and timing are missing. Yet in the real world of neurons, phase and timing are

important and can lead to different results. This gives the biological neuron a whole universe of potential functionality excluded from the ANN formula.

And I haven't started into that sigmoid activation function to the right of the summation in the classic ANN algorithm a few pages back. It has no biological analog at all; it was added on there in the 1980s to make the backpropagation algorithm work to solve some specific problems…I'll return to backpropagation in a moment.

This is not to say that the ANN formula is a *bad* formula. It just doesn't have much to do with biological neurons. Why? The underlying idea of having neurons with analog, continuous values is invalid, and excluding phase and timing eliminates lots of the neuron's potential.

Synapses

Which brings us to synapses for a similar conversation. Once again, the neural network represents the weight of a synapse as a floating-point number, although the neuroscientists tell us that they have a limited number of discrete values. In the previous chapter, I explained how the more values a synapse might take on, the slower the learning process must be.

But let's look at an even more fundamental problem. THERE IS NO WAY TO ACCESS THE WEIGHT OF A SYNAPSE PRECISELY.

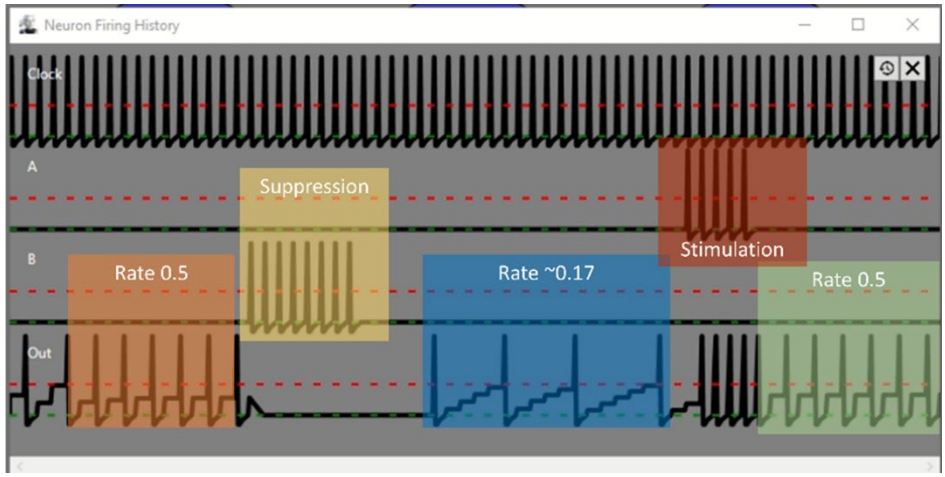
Reconsider the example of a neuron firing at a fixed rate connected to another with a synapse of unknown weight. You observe that the output neuron is firing at half the rate of the input. But that doesn't mean the synapse has a weight of 0.5, it means the synapse weight is somewhere in the range of 0.5 on up to 1.0. How can you tell what the exact value is?

Within the *Brain Simulator*, you can just click on the synapse and read out its weight or click on the Out neuron and see how much the synapse contributes to the membrane potential. But in a biological brain, we don't know how to measure the weight of a synapse and the only way to measure a neuron's membrane potential is with needle electrodes, which is not a very pleasant prospect.

You could imagine a scheme where you fired an input neuron repeatedly until the output neuron fired and count the number of

spikes it took. No consider that if you want to detect a synapse with a weight of 0.01, it will take 100 spikes to get the output to fire. At a firing rate of 250 Hz, this will take nearly half a second; and that was to determine the weight of a single synapse—obviously to slow. If you want to represent a hundred synapse values which is not much precision at all, it will take even more time. So detecting a synapse weight with any degree of precision is impossible, how about setting the weight?

We've all heard that "Neurons which fire together, wire together," which means that connected neurons with near-simultaneous spiking increase a synapse weight while the converse is true.



With B connected with a synapse of weight -1.0 and A connected with a weight of +1.0, you can see how stimulating or suppressing simultaneous spiking will change the weight of the synapse. You also get the idea that setting a synapse to any specific weight is not possible and you can never know, precisely, what the synapse weight is.

With sufficient stimulation, you can be pretty sure that the synapse weight will near 1, and with sufficient suppression, it will approach zero (or -1 or whatever limit values are in the Hebbian formula). In between, though, synapse weights are imprecise. There is no way to set a synapse weight to any specific value like .5 and greater precision is even further out of reach. Finally, there is no practical way for a neuron to discover the precise weight of a synapse.

Worse yet, every time the neurons spike, the synapse weight changes slightly. That means that even if you could set the synapse weight precisely, it won't stay at that weight for very long.

So, if you can't store or read back synapse weights, what good are they? Well, synapse weights are *not* useless, far from it. They're just useless for storing precise values you want to read back. A better way to approach it is that a synapse represents a single bit of information while the weight value represents the confidence that that bit is true—that is, how easily it can be changed.

Once again, looking at the fundamental neural network formula, notice how it relies on the idea of precise synapse weights.

Backpropagation

Which brings us to backpropagation. It represents a family of algorithms that need to be tweaked, trained, and tweaked again. Everyone can give examples of the shortcomings of backpropagation like this one where facial recognition improves if parts of the face are rearranged—but is lost if the image is inverted. Further, you know *your* brain doesn't need thousands of training samples. You can learn a new symbol or a new face in just a few moments…and you're not confused at all if it's upside-down.

By Nick Bourdakos, IBM Watson.

In this example of sub-optimal deep learning, recognition accuracy actually improves when various parts of the face are grotesquely rearranged but is lost altogether if the correct image is upside down.

But that's not my point. Backpropagation *cannot possibly* be representative of how neurons learn…for two fundamental reasons.

First, a quick look at the formula shows that it relies on knowing what current synapse weights are *and* being able to directly modify the weight of any synapse in the network with great precision. This is simply not possible in a biologically plausible world.

$$W_{i,j(new)} = W_{i,j} + \Delta W_{i,j}$$

This excerpt from the backpropagation equations illustrates how backpropagation is completely reliant on being able to accurately read and modify any synapse weight in a system.

Second, the method by which the weight changes are calculated, called "Gradient Descent," will not work if the gradient field is not continuously differentiable…that is if it isn't smooth—which it won't be because of the discrete nature of the neurons and synapse weights in your brain.

Summary

So biological neurons, synapses, and learning aren't like the ANN and I've just scratched the surface. That isn't to say that today's AI

approaches are wrong or don't work. On the contrary, many AI systems work very well. It *does* mean that the algorithms of today's AI are different from the way your brain accomplishes similar tasks because classic ANN algorithms are impossible to implement in neurons. After forty years of experimentation in AI with no emergence of general intelligence, it's time for some new approaches.

The examples in *The Brain Simulator II* illustrate some of the capabilities and limitations of biological neurons and can be used to highlight the distinction between ANNs and real-world intelligence.

Could the *Brain Simulator* support the classic ANN algorithms? Of course. As mentioned in the previous chapters, neuron values and synapse weights are stored internally in floating-point numbers and a Module can be written to have direct access to read and modify any synapse weight in the network. Thus, we could add a neuron model to include the weighted-sum algorithm of ANNs and we could add a Module that implements backpropagation.

Would this be a good idea? Keep in mind that the point of the *Brain Simulator* is to try out new algorithms and experiment with different approaches. The internal structures are designed around the biological neuron which is fundamentally different from the ANN neuron. The biological neuron has internal charge memory, leakage, and timing capabilities while the biological axon can connect synapses to neurons almost anywhere in the brain. For efficiency, the *Brain Simulator* only needs to process neurons when they spike. This is fundamentally different from the ANN, which processes vast arrays of neurons and synapses whether they are active or not. So on the whole, implementing classic ANNs on the Brain Simulator is possible, but not computationally efficient and would likely show that the ANNs work just like the ones on other platforms.

Video Links

"AI is Like Your Brain: DEBUNKED"
http://futureai.guru/videos?id=133

Chapter 4:
Applications of Neurons

This chapter asks you to consider a few things the brain can do within the context of what we know individual neurons can do. Looking at the greatest things the brain can do can cloud the picture. If you understand the workings of the neuron (or a transistor), it's not at all obvious that you can harness many of them to play chess, for example. So let's start with some of the simple things the brain can do.

Obviously, neurons can differentiate between different colors or intensities of light. How? It's not as simple as you might think—actually building a network in the *Brain Simulator* to accomplish even this simple task can be an eye-opening exercise.

We know you can remember things—lots of things for a short time and different things for a longer time. This chapter describes several ways this might happen in neurons.

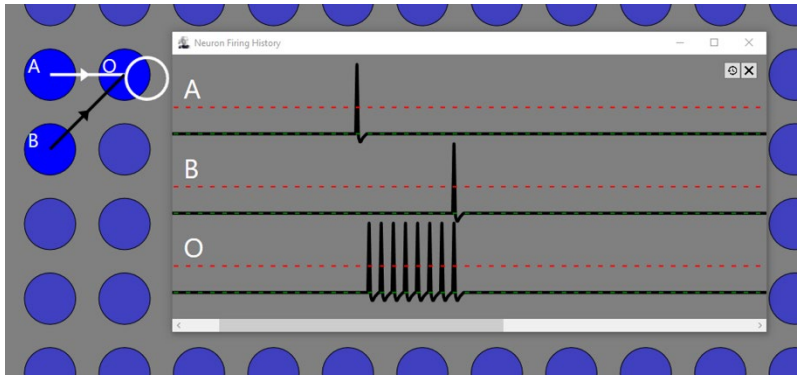
You can determine the direction of sounds with considerable accuracy. How can neurons do that? The speed of sound requires measuring differences of microseconds in sound waves with neurons which take several milliseconds to spike.

The answers to these and many similar questions provide insight into how the brain works and how it could implement general intelligence.

Digital Logic in Neurons

It's worth considering a special case where neuron values and synapse weights are restricted to represent digital circuits, to either 1 or 0. With a synapse weight of 1, any individual neuron will cause its synapse target neurons to spike and, conversely, a neuron will fire if any incoming spike is received. If you consider a neuron with multiple incoming synapses, it acts as an OR gate…it will spike if any incoming signal spikes. Likewise, a neuron that is connected to itself will spike indefinitely if it ever receives an incoming spike. Here's

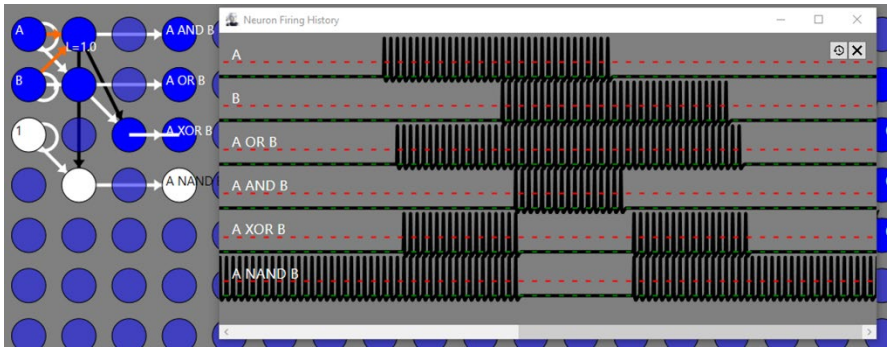
where the -1 synapse comes in. Connect an incoming synapse with a weight of -1 and it will cause the neuron to stop spiking.



Because it is connected to itself with a synapse of weight 1, whenever Neuron O receives a spike from A, it will subsequently spike continuously because it stimulates itself to spike. Subsequently, if it receives a spike from B which is connected with a synapse of weight -1, it will stop spiking.

With this simple network, we've created a single bit of memory. The spiking state of neuron O can represent either a 1 (spiking) or a 0 (non-spiking). There *are* other mechanisms that offer some advantages over this one which will be discussed later.

Appropriate selection of synapses can create networks that implement any basic logic component.



This timing diagram shows you that basic logic functions can be created with simple neurons. As an example, the A OR B neuron spikes when either A or B (or both) is spiking. The A AND B neuron only spikes when both are spiking. This type of logic requires an "always-spiking" neuron (the neuron below B in the array) in order to perform "signal inversion."

While this is potentially useful in its own right, a key feature is to extend this observation to know that given enough neurons, you

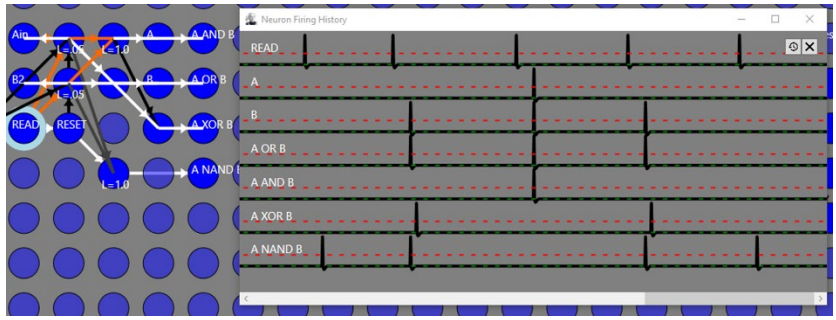
could create any digital circuit. This is because the simple set of logic gates implemented here are "functionally complete" and can be proven to be the building blocks of *any* logic circuit.

For example, consider that the 8086 processor (ca. 1978) contained only 29,000 transistors (at most 10,000 logic gates) and so could be emulated within the *Brain Simulator* with ample neurons left over for other functions. Even this early microprocessor, though, would be perhaps a million times slower if implemented in neurons because computer designs have been optimized around the operating characteristics of transistors while the brain has been optimized around the strengths and limitations of neurons.

The point is to show that even this simple model with synapses restricted to one of two values is sufficient to represent *any* digital circuit. All the complexity of the biological neuron may add some efficiency but as more complex models are described, recall there is no *theoretical* need for them—you could do everything with the IF model and fixed-weight synapses.

Saving Energy

As an alternative, with the slightly more complex LIF model, identical logic functions can be implemented with a different scheme. Instead of continuous firing representing a logic 1, let any single spike represent a 1. This is a bit more difficult to grasp but is much more biologically plausible.



Where the previous demonstration used "always-firing" to represent a logical "1," this uses just a single spike. The logic levels are only valid after the "Read" neuron spikes. When compared with the previous Network, this represents the same logic but the multiple spikes are not there. The great advantage is that this Network requires almost no energy when logic is inactive.

We know that the brain is tremendously efficient. My brain uses only 1/10th the energy of the CPU in my desktop computer. One of

the ways it does this is by not having any neuron spike unnecessarily because spiking requires energy. While this logic family is just a bit trickier, it forms an equally functionally complete set.

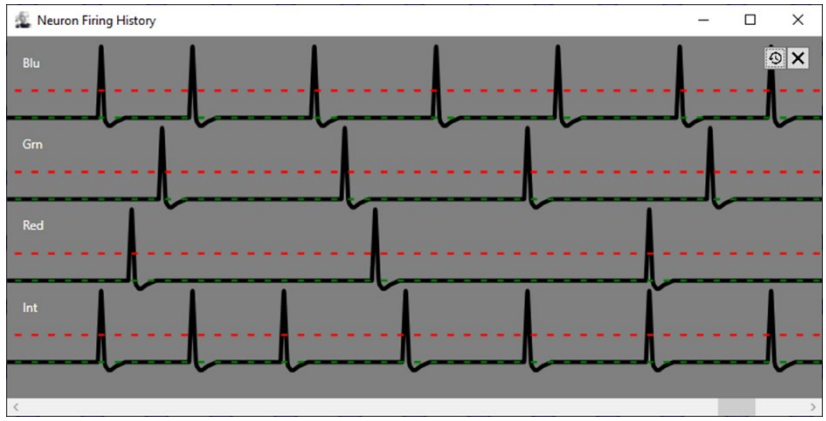
It's difficult to know if your brain uses logic like this. If you were probing a brain looking for activity, the always-firing logic would be immediately obvious while the single-spike logic would not be. Single spikes could be performing a variety of logic functions in the brain and we will not be able to determine what they are until we develop the technology for tracing individual connections.

Frequency/Rate Detection

Sensory signals which arrive at the brain contain information in the form of the spiking rate. Whether it is color, sound intensity, touch, etc., the stronger the sense, the faster the relevant neuron will be spiking. Similarly, the brain's output to muscles, most of the actions your brain can perform, cause muscles to contract more strongly with faster spiking.

Rate-based signals at the inputs and outputs of the brain have led many to presume that all the internal processing of the brain is likewise rate-based. But when you consider the types of things your brain needs to do, though, you can convince yourself that interior signals of the brain represent meaning in individual neural spikes or clusters of redundant spikes.

Consider how the brain might answer these questions: What color is this point? Are these two adjacent points the same color (is there a boundary)? The answer to both of these is that neurons need to be able to compare the firing rates of different neurons.

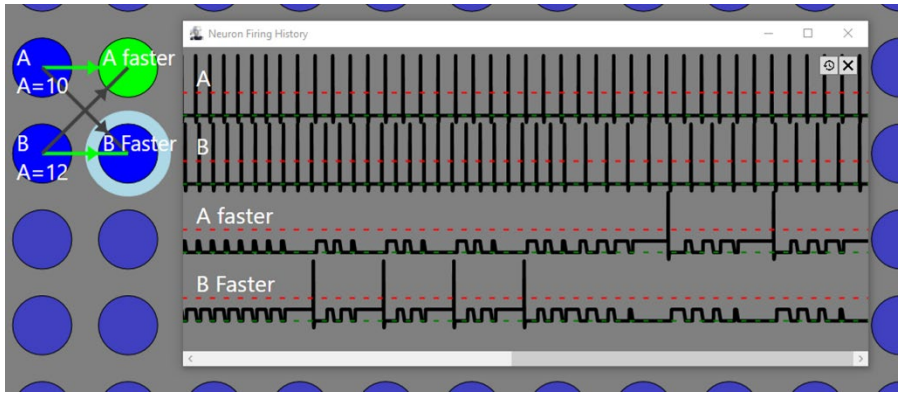


This timing diagram, which might represent spikes coming from the retina, illustrates the difficulty of identifying different colors. Your brain gets signals like these; can you tell which color it represents? This timing is created with a Module that can read pixels from a video camera and generate the spiking rates for Blu, Grn, Red, and overall intensity neurons. A small amount of randomness is introduced to prevent the signals from synchronizing.

Different neurons representing blue, green, red, and overall brightness all spike asynchronously at different rates. Any specific color could be identified by detecting specific rates of three neurons (Blu, Red, Grn) simultaneously while a boundary might be located by detecting different firing rates from two adjacent intensity neurons.

There are two ways to look at any rate-based signal. The first would be to count the number of spikes in a given timeframe. The other would be to measure the time between any adjacent pair of spikes. The former is more immune to noise because it averages over a longer timeframe but the latter can generate a result more quickly (for faster signals).

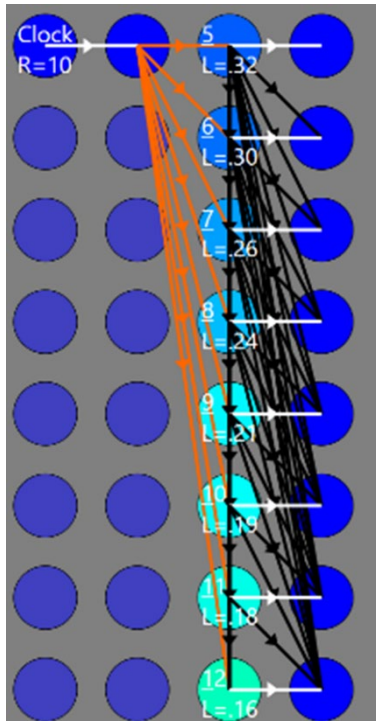
A simple way to detect which of two signals is firing faster is to connect them to each of two different neurons with synapses of weight 0.5 and -.5. The neuron with the faster-spiking signal will accumulate charge faster in one and cause it to spike while suppressing the other.



A is spiking at rate 10. On the left side of this timing window, B is spiking at a rate of 11 so A is faster. Midway through the display, B's rate is reduced to 9 so A is faster.

On closer inspection, this circuit relies on the fact that a faster-spiking neuron will, at some point, fire two spikes in a period where the other fires none. There are two problems with this simplistic approach. First, any noise in the two signals will keep it from working properly. Second, and perhaps more important, it's slow. If we assume a cycle time of 1 ms, it takes 100 ms for this circuit to determine which signal is faster because spikes come about 10 ms apart and it may take 10 spikes to detect the difference in rate. Imagine how slow your brain would be if it took a full tenth of a second to determine even the simplest difference. So we need a more sophisticated solution. It will take more neurons but will be much faster.

The following network can discriminate between eight different spike rates but could be extended to an arbitrary number (within time and noise constraints). To detect a visible boundary, two of these networks would be connected to two adjacent pixel intensity signals and the outputs are ANDed so that if the same level is detected on both, no boundary exists. The second logic example in the previous section could be used because it has the side-effect of handling incoming spikes which are not synchronized.



A Network like this can detect eight different rates of incoming signal. Neurons in the center column have different leak rates so they act as high-pass filters with different cut-off frequencies. Neurons in the right-column have synapses that limit firing to only one specific frequency at a time. In this example, the refractory period is 4 and the signals detected range from 5-12 cycles between spikes. This circuit registers the rate on every other spike.

Color detection requires three such discriminators. Similarly, recognized colors indicate neurons which are the AND of three specific red, green, and blue levels. Only eight levels may be needed for each color (yielding 512 recognizably different colors) while more levels would be needed for boundary detection. This would account for the optical effect that two objects may appear to be the same color until they are next to each other so the boundary can be detected. Similarly, because the specific location of a boundary is important while a color is a property of an area, we can expect many more intensity (gray-level) signals than color signals.

Four Memory Mechanisms

The question is: given what we know about how neurons work, how can they be harnessed to store information?

In the same way, the computer storage is hierarchical with CPU, RAM, and SSDs or disks all contributing with trade-offs of speed, cost, energy consumption, and permanence, memory in the brain has different needs for different parts of the thinking process. For example, your brain needs to store an object's position for a very short time in order to know if it is moving. It doesn't matter how many neurons it takes (cost and energy), but you only need this

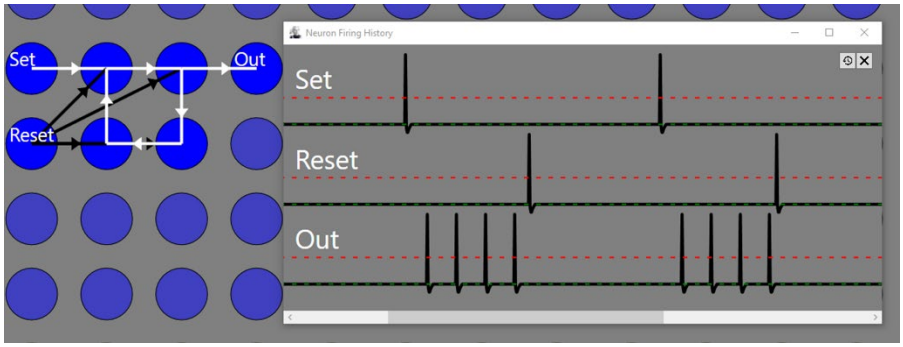
memory for a fraction of a second. At the other end of the spectrum, you have long-term memories (like childhood memories), and these need to be essentially permanent but consume no energy to maintain the memory.

I see four ways that neurons can store information and others may come to light. I have already touched on one method of using neurons to store bits of information by firing continuously, and here I add three more. Because of the size and varied structure within the brain, it is likely that all of these are used.

Memory in Spiking State

Previously, I described how a neuron connected to itself with a synapse of weight 1.0 can act as a single bit of memory. It will begin to fire if it is stimulated and will fire continuously until it is suppressed. This has the advantage of being easy to explain, requires one neuron per bit, and is very fast (for a neuron). But we need to consider that spiking neurons consume energy. This means that storing information in continuous spiking is unlikely to be a widespread approach in the brain.

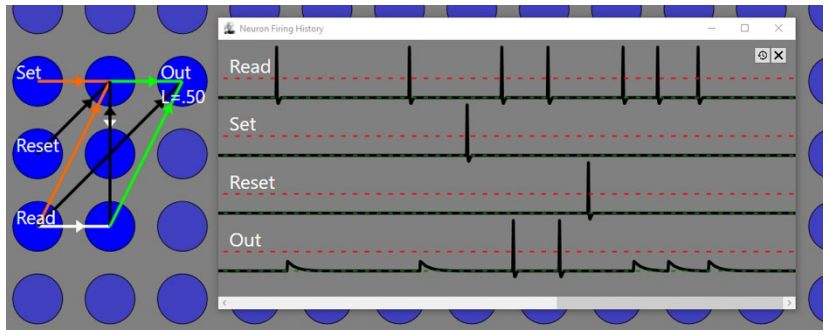
As initially described, this method oversimplifies the problem with a neuron synaptically connected to itself. When such a neuron emits a spike, it will likely be received within the neuron's refractory period and be ignored. This can be overcome by increasing the axon delay or by having multiple neurons forming a ring of connections (as shown).



With a refractory period of 4, a ring of four neurons, each firing the next, stores a single bit in the firing state. The ring starts firing on Set and stops on Reset.

Memory in Internal Charge State

Let's consider storing information in the internal charge state of a neuron. For example, a neuron represents a logical 1 if the internal charge is 0.1 or greater and a logical 0 otherwise.



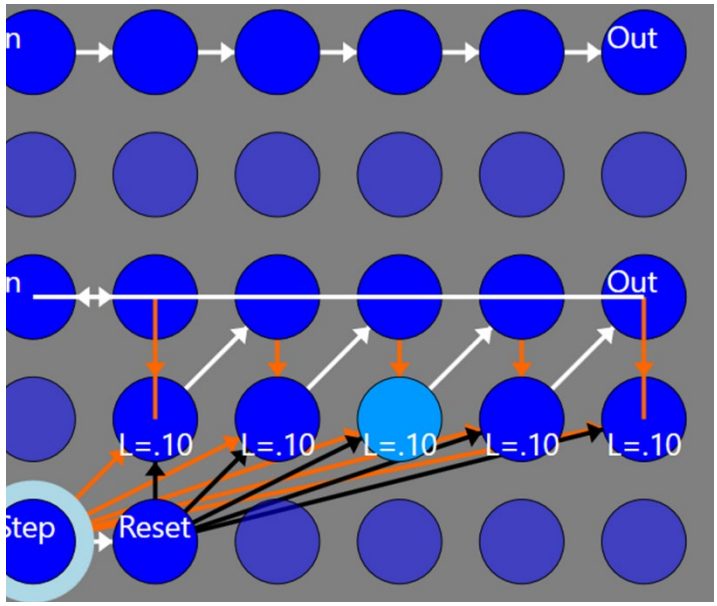
In this circuit, when "Read" spikes, the content of the memory will appear on Out. After "Set" has spiked, Out will spike after Read. After "Reset" has spiked, Out will NOT spike after Read. The two unlabeled neurons in the center constitute the memory bit while the other neurons can be common to any number of bits.

This idea requires at least two neurons per bit because the "Read" operation is "destructive". That is, in order to read the internal state of a neuron, you must alter its internal state, possibly causing it to spike, but then you need to restore the original state back into the neuron. This memory has very fast store and retrieval times and uses no energy when it is not being read or modified.

Using an IF model, the storage time is infinite but with a more realistic LIF model, the internal charge will decay so the memory must be refreshed by reading periodically. Interestingly, computer DRAM has exactly the same issue (that the charge state decays with leakage) and needs to be refreshed for the same reason. If the memory is not refreshed, it will gradually lose its content. On the other hand, this may be a useful mechanism for short-term memory in the brain. That is, the memory is never reset but only stored and read. Simply waiting some amount of time (perhaps a second) is sufficient to clear the memory by leakage.

Memory in Shifters

Another candidate for short-term or intermediate-term memory is the shifter or delay line.



Two types of shifter are shown. The upper transfers a spike directly from one neuron to the next and so is a fixed time delay. The lower advances the spike by one neuron every time the Step neuron fires and so can provide a variable amount of delay.

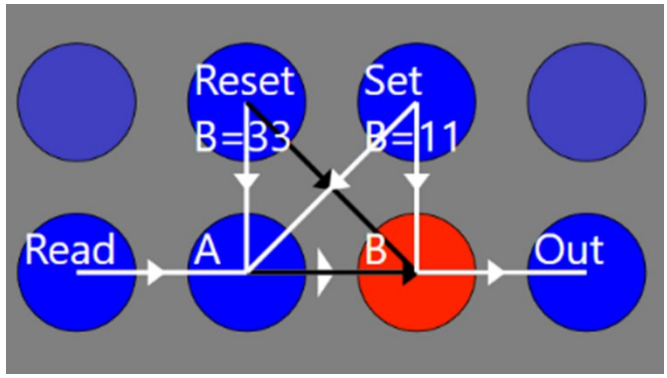
A delay line can be thought of as a bucket-brigade where the incoming signal, "In", is transferred down a chain of neurons. The length of the delay-line limits the amount of memory. Synapses could be added at any intermediate step so, for example, you might think back in your short-term memory to recall the last word, or the last phrase, or more.

The delay line requires one or two neurons per bit depending on whether speed control is needed. As an example, short-term memory for incoming audible signals might incorporate the simpler mechanism while the process for creating speech might use the more complex design so that you can speak at whatever speed you like.

Memory in Synapses

The classic AI mechanism is storing information in the weights of synapses—you could think of a synapse of weight 1 as representing a 1 and a synapse of weight 0 representing a 0. This memory mechanism offers the advantages of being able to store much more data as there can be thousands of synapses for each neuron. Further,

this is the only memory mechanism with any degree of permanence. If you want data to be stored for days or years, this is the mechanism for you because the previous mechanisms either use too much energy or decay over time. The disadvantage is that this mechanism is much slower to change, requiring many spikes to set the synapse weight.



The idea of storing data in a synapse is not as simple as it sounds. The single Hebbian synapse from A to B can be set to a weight of 1 or 0 by the Reset and Set neurons which each fire bursts of spikes sufficient to fully change the weight. Shorter bursts could set the synapse to an intermediate value but making use of some intermediate value is more complex.

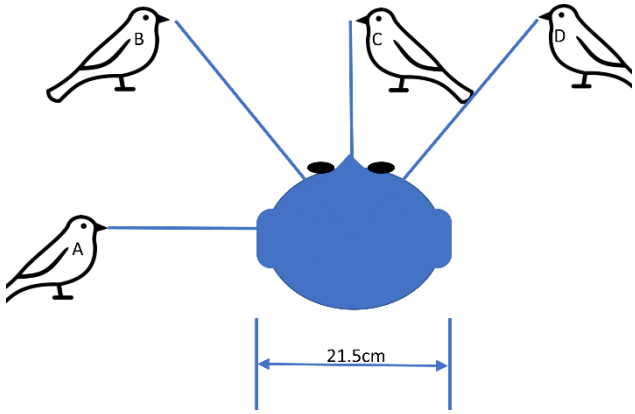
You might think the analog nature of a synapse might let you store any value in the synapse weight as is the classic idea of ANNs. But experimentation with *Brain Simulator* leads to the conclusion that although you might set the weight of a synapse to one of several possible values, there is no practical way to read back the weight without using multiple additional neurons—nullifying the advantage of storing information in synapse weights.

Axon Delays

Neurons can be used to handle timing far more accurately than you might think. Even though a neuron can only fire once every 4 ms and the neural spike is 1 ms long, the neuron (both biological and simulated) can differentiate very small timing differences through a neat trick.

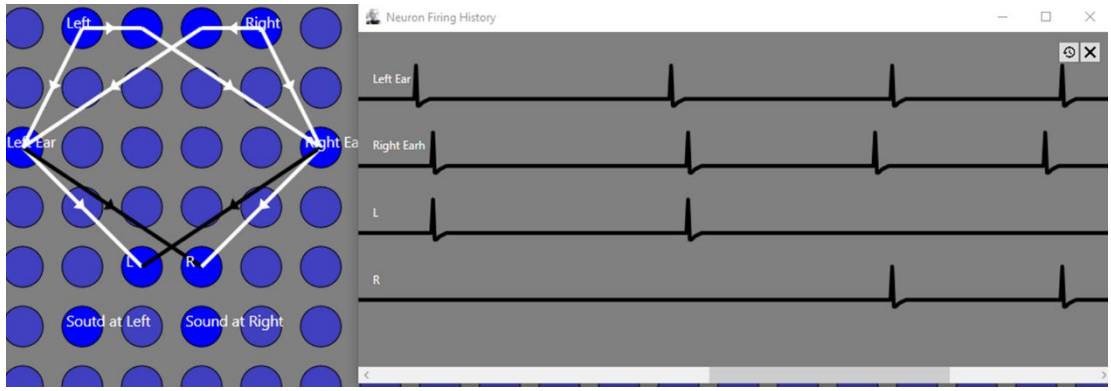
For example, when you hear a sound, you can tell which direction it came from with reasonable accuracy. One of the important directional cues is that the onset of the sound arrives at your two

ears at different times depending on the direction. A sound directly in front of you will arrive at both ears simultaneously while a sound from the side will arrive at the nearer ear first. Different angles will yield different delays.



One way you can localize sound is by detecting the difference in the time of arrival of sound at your two ears. The chirp from Bird C will be heard at the same time in both ears. A chirp from Bird A will be received at your left ear 0.62 ms before it arrives at your right ear. The sounds from birds B and D will arrive at your ears only 0.40 ms apart.

With the speed of sound at 343 m/s and the distance between your ears at 21.5 cm, the maximum time difference is only 0.6 ms. A sound at a 45-degree angle will have a time difference of 0.4 ms. How can the slow neurons in your brain detect the difference between 0.6 ms and 0.4 ms? The answer sheds light on why timing is important in neural modeling. We can be sure that timing is important in this specific instance, but we don't know that it plays an important role in general intelligence.



The neural circuit demonstrates how your brain could localize sound by detecting whether your left or right ear receives a sound signal first. In this example, L fires only if the left ear receives the signal first and R fires only if the right ear receives the signal first. To create accurate localization, sub-millisecond timing is needed so we would need multiple L and R neurons for different angles and the synapses leading to them from the ears need different Axon Delays.

Consider that if a neuron receives a +1 and -1 simultaneously or if the -1 arrives first, it will not spike (as in the previous models). If the -1 comes later (even by a tiny amount once the spike is being emitted), the -1 signal will arrive in the refractory period and will be ignored. How can your brain control the arrival times so precisely? With varying axon lengths. By having multiple neurons connected to both ears with varying axon lengths, your brain can determine the angle with great precision. In the same way that the previous rate-detector had different neurons which responded to different spike timings, an array of neurons could fire only for specific sound directions. In order to simulate this in the brain simulator, you'd need to set the refractory period to 40 or even more so you could represent timing with sub-millisecond precision.

Video Links

"How Your Brain Works: Part 2 Neurons"

http://futureai.guru/videos?id=107

"Short: Neurons"

http://futureai.guru/videos?id=139

"Short: Single-Spike Conversion"
http://futureai.guru/videos?id=138

"Short: Short-Term Memory with Neurons"
http://futureai.guru/videos?id=137

Chapter 5: Networks

The combination of neurons and synapses forms a "network" that might perform some interesting or useful function. Every neuron and synapse within the network has some specific state depending on its model (as described previously) which defines what will happen next.

Networks can be saved to files and restored for future computation. At present, the complete state of the network is stored in an XML file (a standard document file format) which can be examined with any basic text editor like Notepad. When the file is reloaded, the processing can continue exactly where it left off when the file was saved.

The network files can be thought of as being like document files with *Brain Simulator II* being the application that edits them. The file is created with a Save or SaveAs function and then can be opened, edited (or run), and saved again. You can have any number of network files and network files can have any number of neurons and synapses.

Also, like documents (as described in Chapter 7 on the User Interface), portions of the network can be copied, pasted, or stored to new network files in their own right. That means that if a portion of one network does something useful, it can be copied and included in another network. Further, portions of a network can be repeated within the network with multiple paste operations.

Network files also contain a "Notes" section, which allows for a description of what the network does and how it might be used. When a network is first opened, the Notes will be displayed in a read-only dialog.

The concept that an overall AGI network will be created from numerous instances drawn from a library of neural functionality is a key theory in the architecture of the system. It is a reasonable idea based on the knowledge that the brain contains many more neurons and synapses than could be explicitly described (or even initialized)

in our DNA. Instead, it is likely that our DNA defines basic network structures and these structures grow and repeat within the brain as it develops.

This is an intriguing field of future research as our DNA defines chemistry and how it might define a brain is unknown. In AGI, many people presume that the human brain emerges largely devoid of content, but consider a horse that is born with the ability to walk, see, avoid obstacles, and a host of other functions which are immensely complex. How the neural connections needed for these functions are defined by the horse's DNA remains a mystery.

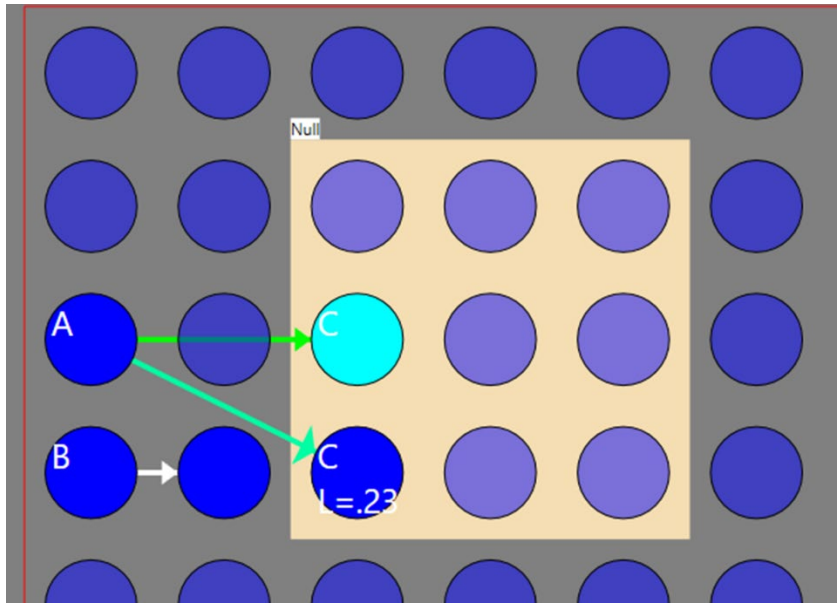
What's in a Network File

There is no need to know the internal structure of a network XML file, but knowing what information is included sheds some light on the information necessary for *Brain Simulator* operation. If you *are* familiar with the format of XML files, you'll find fairly straightforward content.

The network file includes the complete content and state of the network. The intent is that if you are running a network, you can stop the engine and save it to a file. At some future date, you can reload the file and continue the execution of the network exactly where it was left off.

So what's in a network?

- State of all Neurons: Model, label, internal charge, and other model-dependent parameters.
- State of all Synapses: weight, target neuron, Model.
- State of the User Interface: Notes, scale, position, engine state.
- State of all Modules: label, Location, function, internal state as defined by the module itself, any module dialog display.



This trivial demo Network shows how the Network content is represented in an XML file. The intent is to show the straightforward content of the file and give an idea of how the representation of millions of neurons and synapses might result in a very large file. It is not expected that these files would be edited outside of the Brain Simulator.

```
<?xml version="1.0"?>
<NeuronArray             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <networkNotes>Demo Notes</networkNotes>
  <hideNotes>false</hideNotes>
  <Generation>1645</Generation>
  <EngineSpeed>250</EngineSpeed>
  <arraySize>450</arraySize>
  <rows>15</rows>
  <lastFireCount>0</lastFireCount>
  <displayParams>
    <NeuronDisplaySize>62</NeuronDisplaySize>
    <DisplayOffset>
      <X>0</X>
      <Y>0</Y>
    </DisplayOffset>
    <NeuronRows>15</NeuronRows>
  </displayParams>
  <ShowSynapses>true</ShowSynapses>
  <Modules>
    <ModuleView>
      <AreaNumber>0</AreaNumber>
      <Label>ModuleNull</Label>
      <FirstNeuron>31</FirstNeuron>
      <Height>3</Height>
      <Width>3</Width>
      <Color>16113331</Color>
      <CommandLine>ModuleNull</CommandLine>
      <TheModule xsi:type="ModuleNull">
        <initialized>true</initialized>
        <dlgPos>
          <X>0</X>
          <Y>0</Y>
        </dlgPos>
        <dlgSize>
          <X>0</X>
          <Y>0</Y>
        </dlgSize>
        <dlgIsOpen>false</dlgIsOpen>
      </TheModule>
    </ModuleView>
  </Modules>
```

```
<Neurons>
    <Neuron>
      <Id>2</Id>
      <Label>A</Label>
      <Synapses>
        <Synapse>
          <Weight>0.9</Weight>
          <TargetNeuron>32</TargetNeuron>
        </Synapse>
        <Synapse>
          <Weight>0.34</Weight>
          <TargetNeuron>33</TargetNeuron>
        </Synapse>
      </Synapses>
    </Neuron>
    <Neuron>
      <Id>3</Id>
      <Label>B</Label>
      <Synapses>
        <Synapse>
          <TargetNeuron>18</TargetNeuron>
        </Synapse>
      </Synapses>
    </Neuron>
    <Neuron>
      <Id>32</Id>
      <LastCharge>0.25</LastCharge>
      <Label>C</Label>
    </Neuron>
    <Neuron>
      <Id>33</Id>
      <Model>LIF</Model>
      <LeakRate>0.23</LeakRate>
      <Label>C</Label>
    </Neuron>
  </Neurons>
</NeuronArray>
```

This listing of the simple demo Network XML file shows the content of a Network. In the left column (the first part of the file) are the Network notes and various display parameters. This is followed by the list of Modules. Each Module may store different information and some Modules (like the UKS described in its own chapter) may have thousands of lines of content.

Lastly (shown in the right column), the neuron array. To save space, the file excludes unused neurons and any values which are at default levels. Each neuron is identified by its ID, a label if it has one, and various other parameters. This is followed by the list of synapses, each of which has a weight and a target neuron. In the right column, note that the neuron with ID=2 has a label of "A" and two synapses, the first, connecting to neuron 32 with a weight of 0.9 and the second, to neuron 33 with a weight of 0.34. At the bottom of the column, Neuron ID=22, labeled "C," is an LIF model with a leak rate of 0.23. Neuron ID=32 is the only neuron in the Network not at its resting potential and this is indicated by the presence of the LastCharge=0.25 line.

At the time of writing, the network file does not include information about multiple-server configurations. A network running across multiple servers is stored as a single large network. There is currently no provision for loading a file into a multiple-server setup.

The Clipboard

Chapter 7 describes how to use the clipboard to cut, copy, paste, etc. selected areas of a network. The key feature is to know that the clipboard content represents a Network in its own right. If you store the content of the clipboard to a file, it is stored in a format identical to a full network. Not only can you then load the file back into the clipboard for inclusion into another network, but you can open the stored clipboard content directly, edit it, and save it again first.

List of Current Networks (v1.0)

These are the networks that are distributed with the *Brain Simulator* download.

BasicNeurons.xml—illustrates the simplest neuron models.

CameraTest—inputs the camera on your computer to neuron values.

SpeechTest—uses speech recognition and synthesis.

SimVision—shows Sallie's simulated environment and vision.

Imagination—shows how Sallie's internal mental model can be used for imagination.

BabyTalk—learns to speak by trial-and-error learning. Where SpeechTest works with words, BabyTalk works with phonemes.

Maze—shows navigation of a maze the way a 3-year-old might. As Sallie explores the maze, she remembers landmarks, the decisions she made, and the results which were achieved. These are the necessary components of reinforcement learning.

Sallie—is an end-to-end AGI model using hearing, vision, and knowledge to learn the meanings of a few words.

3DSim—demonstrates the 3D simulator under development.

NeuralGraph—demonstrates how a mathematical graph can be implemented in neurons controlled by a module.

ObjectMotion—demonstrates how Sallie can move objects in her environment.

Chapter 6: Modules

So far, I have described the functional network of neurons and synapses so now we need to address how a network can be created. In Chapter 7, I'll describe how this can be done by hand, but this approach is only applicable to small functions. On larger functions, the "by-hand" approach becomes immensely tedious.

Enter the "Module" —a powerful addition to the simulation process.

Any rectangular cluster of neurons can be assigned to be a Module and a Module is backed by computer code in a high-level language. All Modules to date are written in C# but could be in any language supported by .NET, including C++ and Python. Modules have direct access to all the underlying resources of the simulator, including things like adding, deleting, or modifying synapses, or reading or changing the values of neurons. Because the code within the module has full control of the network, there is no limit to the functionality that is possible.

Let's start with the basics. The Module has two primary methods: "Initialize" and "Fire". The Initialize method is executed only once when the Module is first added to a network or if requested by the user. The Fire function is executed once for each cycle of the Neuron Engine. Within the Initialize method, a Module might allocate a slew of synapses. These synapses can not only connect to neurons within the module but can connect to or from any neuron in the network, including neurons in other modules. Both neurons and Modules can also be referenced by label. Modules can also access the characteristics of other Modules. In this way, for example, a Module performing some vision function can set its dimensions as appropriate to the size of the input image and create synapses connecting the input image to its neurons.

While all neural functions could theoretically be created in synapses, there are many which are much more convenient to

implement in code. Again, thinking of some vision function, instead of creating a slew of neurons in the Initialize method (to perform the function), the Fire method can sample the state of the neurons in the input image and set values for its own neurons directly—eliminating the need to perform the function with neurons and synapses altogether.

```
//
// Copyright (c) Charles Simon. All rights reserved.
// Licensed under the MIT License. See LICENSE file in the project root for full license information.
//

namespace BrainSimulator.Modules
{
    public class ModuleNull : ModuleBase
    {
        //Any public variable you create here will automatically be stored with the network
        //unless you precede it with the [XmlIgnore] directive like this
        //[XlmIgnore]
        //public theStatus = 1;
        public bool doWork = true; //this will be saved to the xml file because it's public

        //Fll this method in with code which will execute
        //once for each cycle of the engine
        public override void Fire()
        {
            base.Fire();  //be sure to leave this here
            Neuron n = GetNeuron("FirstNeuron");
            if (n != null)
            {
                if (n.CurrentCharge > 0.5f)
                    n.SetValue(1);
            }
        }

        //Fill this method in with code which will execute once
        //when the module is added, when "initialize" is selected from the context menu,
        //or when the engine restart button is pressed
        public override void Initialize()
        {
            Neuron n = na.GetNeuronAt(0, 0);
            n.Label = "FirstNeuron";
            Neuron n1 = na.GetNeuronAt(0, 1);
            n.AddSynapse(n1.Id, 0.5f);
        }

        public override void SetUpAfterLoad()
        {} //add code here to execute once after the module is loaded from a file
        public override void SetUpBeforeSave()
        {} //add code here to execute once before the module is saved to a file

    }
}
```

This c# code demonstration Module program shows the overall structure and simplicity of a Module. Within the "Fire" method, the program finds a neuron by label and if its charge is greater than 0.5, it sets it to 1 (which fires it). Within the "Initialize" method you can see how to locate a neuron by its position, add a label, and add a synapse to another neuron.

Further, within its Fire method, a Module might send or receive signals to other functionality within the computer. For example, a robotic Module might sample some neuron values and then send the appropriate signals to various robotic servos to perform some action. The previously mentioned input image might accept input from a video camera and set neuron values as appropriate. Alternatively, the input image could be read from any image file. The huge breadth of opportunity is detailed in some of the Modules described in subsequent chapters.

The layout and content of Modules are included in the network file when it is saved. This means that the Module might have some internal state and this is automatically saved and restored with the network. As an example, a Module such as a world simulator can create a set of obstacles in code and these will be saved and restored automatically. So, if an AGI moves an object in the simulator, it will stay moved for subsequent runs.

The ability to put any code into a Module means that instead of using neurons, *any* functionality can be created in software. Modules may also reference each other's methods directly but this idea is being phased out in favor of always interfacing through neuron values.

From a programming perspective, any variables within a Module declared as "Public" will automatically be saved and restored in the XML network file unless explicitly excluded with an "[XmlIgnore]" directive.

A Module might also have a dialog box. Again, using the example of the simulator, the dialog box can show the locations of the AGI in the simulation along with the positions of all the other obstacles within the simulated world. Other modules can have dialogs that display text content…for example, the speech recognition or text-analysis modules.

At this point, all Modules are single-threaded and run sequentially in each engine cycle. They are processed in the order of the ID of their upper-left neuron. This may be changed in the future and should not be relied upon.

Using Modules for Interfaces to the World

Obviously, individual neurons of the *Brain Simulator* can't access a camera or microphone for input or control a robot for output because all they can do is accumulate synaptic inputs and emit spikes. Instead, simple Modules can perform these functions.

A Module with just a few lines of code can access a video camera and put the content into neurons which can be used for other processing. Conveniently, another Module can read images from files and put them into the same neurons for more repeatable downstream processing.

Another Module handles incoming speech. While it would also be easy to create Modules that read raw microphone input, perform signal processing, and do speech recognition, this wheel has already been invented. Instead, the SpeechIn Module uses the operating system's intrinsic speech recognition engine. This can be used at the level of firing neurons which represent individual words or at a lower level where neurons represent individual phonemes. For output, Modules can convert neural pulses to servo controls for robotics or speech output (again via the operating system).

Finally, a simulator Module can simulate the functions of all the sensory and output functions. The advantage of a simulator is that the inputs can be simple and repeatable. Real-world visual and audio input is immensely difficult to process and within the simulator, you can make things as simple as you like—then rerun the exact same input to debug other areas of the network.

Using Modules for Computational Efficiency

One key argument that AGI is coming sooner than most people think is that there are numerous functions that a computer can perform much more efficiently than any array of neurons.

Let's consider a few examples. Consider the very simple networks described previously which perform logic functions. While it's possible to perform these functions in neurons, they will be millions of times faster in a few lines of code within a Module.

Next, consider that you have a sequence of actions you'd like to perform and that individual neurons can perform each individual action. You'd like your network to learn a "macro" that would fire

the individual actions in order. One way to do this is touched upon in the delay line described in the previous chapter. Each step in the delay line can learn to fire the appropriate action neuron in sequence. You can do this with a minimum of two neurons per output step plus the neurons needed to learn the sequence. This cumbersome process is likely what the 56 billion neurons of the cerebellum are doing.

The reason this is cumbersome in neurons is two-fold. First, neurons in the brain don't have specific addresses and are not accessible in a specific order. Second, all of the synaptic signals from a neuron arrive at their targets at essentially the same time. The CPU has a significant advantage because computer memory is inherently sequential. The CPU can access the *Next* item in RAM because the concept of *Next* is defined by the CPU's addressing space. This is not the case with neurons. Neurons appear to be accessible only by the configuration of their synapses, that is, the content they represent. Also, within the simulator each neuron maintains a list of its synapses and the neurons they target. In a biologically plausible world, all these are processed simultaneously. But in a computer, it's a simple matter to direct that synapses be processed sequentially. This is illustrated in the Universal Knowledge Store Module described later.

Using Modules for Functions That are Difficult in Neurons

We know that your binocular vision can use the differences in the images presented by your two eyes to estimate the distances to objects you see. This is the basis for the illusion created by 3D movies. I don't know how the brain accomplishes this task but it is reasonable to assume that it is complicated. Because you know where things are in your immediate surroundings, even with your eyes closed, this estimated distance is important for processing that occurs downstream from object recognition.

Rather than letting development be blocked by this problem, I wrote a Module that uses trigonometry to estimate visual distances. There is no reason to think this approach has any relation to the way your brain works but it accomplishes a similar goal. With this estimated distance information, we can continue to experiment with and develop the mind's internal model of its surroundings. In future

development, this Module might be replaced with a more biologically plausible approach. Alternatively, we might conclude that the trigonometry approach is significantly more efficient than the way your brain works and the Module might continue to be used for computational efficiency.

List of Current Modules (v1.0)

Some Modules have a custom dialog which can be displayed to edit the Module's parameters while the Neuron Engine is running. This is indicated by the text "Has Dialog" after the Module name in the following list.

Module2DModel: (Has Dialog) Manages the content of the UKS to create persistent memory of Sallie's two-dimensional surroundings. It automatically updates positions so they are correct relative to Sallie's current position and orientation. Each object position has an associated confidence level (based on the accuracy of the distance estimate) and this is represented in the dialog by the length of white ends on segments. By temporarily adding segments or changing Sallie's perceived position, Sallie can "imagine" surroundings with new objects or from a different point of view.

Module2DSim: (Has dialog) Maintains Sallie's simulated surroundings. Sallie's position and orientation are maintained from Move and Turn Modules and directly output to Sallie's various sensory Modules. Detects collisions between Sallie and objects and moves objects based on assumptions of center of mass and friction.

Module2DSmell: Sallie's limited sense of smell. This Module has two rows of neurons representing input from two aroma sensors which is the strength of a field from green objects. Within the simulator, only green objects have an aroma. It receives input directly from the Module2DSim Module.

Module2DTouch: When one of Sallie's arms contacts a simulated object, this Module fires neurons indicating the position and angle of the touch and whether or not the touch was at the end of an object. This can update information in the 2DModel since the distance value of touch is much more accurate than visual depth perception.

Module2DVision: Updates information in the Module2DModel based on the content of the current field of view. Uses binocular

information from Module2DSim to estimate distances using trigonometry.

Module3DSim: (Has Dialog) Allows Sallie to move about in a three-dimensional world. Only Sallie's visual input is shown in the dialog display.

ModuleArm: Allows for control of Sallie's individual arm positions in Module2DSim. Each instance of the Module controls one arm. This forms the basis for Sallie's ability to explore objects by touch along with Module2DTouch.

ModuleAudible: Works with the UKS to manage Phonemes, Words, and Phrases. This is analogous to the Module2DModel in that it manages UKS content related to Sallie's surroundings.

ModuleBase: (Has Dialog) This is the Base Class from which all other modules are derived. Useful only from the programming interface.

ModuleBehavior: This is somewhat analogous to the brain's cerebellum in that it can manage sequences of primitive physical behaviors. For example, to turn or move a specific amount, multiple smaller moves or turns may be required.

ModuleBoundary: Works with ModuleImageFile to find visual boundaries.

ModuleCamera: Analogous to a retina. Takes input from an attached video camera and sets neuron values to represent the colors seen at specific locations.

ModuleColorComponent: Converts a neuron with the Color model into neurons that have firing rates appropriate to the RGB and brightness components of the color.

ModuleCommand: (Has Dialog) Can read, edit, and execute test scripts. Each step can fire any labeled neurons in any Module by name and can test for (and wait for) results.

ModuleEvent: Works with the UKS to manage memory of events, actions, and outcomes so that Sallie can learn which behaviors are best in various situations.

ModuleFireOldest: This will fire the neuron within the Module which fired the longest ago. This could be useful in selecting things to forget—if a neuron hasn't fired in a long time, it possibly doesn't contain useful information.

ModuleGoToDest: This Module demonstrates the use of imagination in determining a route. The Module works with the 2D model to imagine the world from a different (remembered) point of view.

ModuleGraph: This predecessor to ModuleUKS implements parent/child, next, and other relationships in neurons.

ModuleGrayScale: This Module works with ModuleImageFile module to generate a grayscale image from the component color values.

ModuleHearWords: This Module works with ModuleUKS to manage word and phrase storage.

ModuleImageFile: (Has Dialog) This Module reads an image file in BMP or PNG format and sets color neuron values as appropriate. Optionally, it will cycle sequentially through all the image files in a directory, loading them one at a time.

ModuleKBDebug: (Has Dialog) This Module records neuron firings in and out of the ModuleUKSN to create a transaction display.

ModuleLife: This allocates synapses to make an array of neurons act to simulate Conway's *Game of Life*.

ModuleLineFinder: This Module works with the Boundary Module to find linear sections of a boundary. Future development will create ModuleArcFinder.

ModuleMotor: Analogous to the brain's motor cortex. This Module consolidates Move and Turn functions.

ModuleMove: This Module distributes motion required to Module2DSim, Module2DModel, Module3DVision, and Module2DVision.

ModuleMoveObject: This Module works with Module2DModel to allow Sallie to create a sequence of actions to achieve a goal. She first moves an object to learn how her pushing on it causes it to move or rotate and then moves the object to a goal location.

ModuleNavigate: This Module works in the 2D environment using ModuleUKS to allow Sallie to solve mazes using landmark memory.

ModuleNull: This Module does nothing. It contains a small amount of demonstration code to show how neurons and synapses can be manipulated.

ModuleSpeakPhonemes: (Has Dialog) This Module works with ModuleUKS to learn words in terms of underlying Phonemes.

ModuleSpeakWords: Uses the Windows speech synthesizer to create speech with ModuleUKS.

ModuleSpeechIn: Uses the Windows speech recognition system to create neuron firings from speech.

ModuleSpeechOut: Uses the Windows speech synthesizer to create speech from neuron firings.

ModuleStrokeFinder: (Has Dialog) Along with ModuleLineFinder locates strokes within an image. A "stroke" is the center between two parallel boundaries.

ModuleTurn: Distributes Sallie's rotation to modules that need the information: Module2DSim, Module3DSim, Module2DModel, and Module2DVision.

ModuleUKS: (Has Dialog) The abstract Universal Knowledge Store. (See Chapter 10).

ModuleUKSN: The Universal Knowledge Store with the addition of a neuron interface.

Chapter 7:
The User Interface

One thing which sets the *Brain Simulator II* apart from other neural simulators is its user interface, which includes a display of the content of the simulator. Rather than a black box that simply displays an answer, the neuron content and Module dialogs can show exactly what is going on as the network evolves. The main thrust of the *Brain Simulator* is to create AGI and, along the way, various pieces need to fit together and coordinate to create a whole. Being able to see the pieces is a key feature.

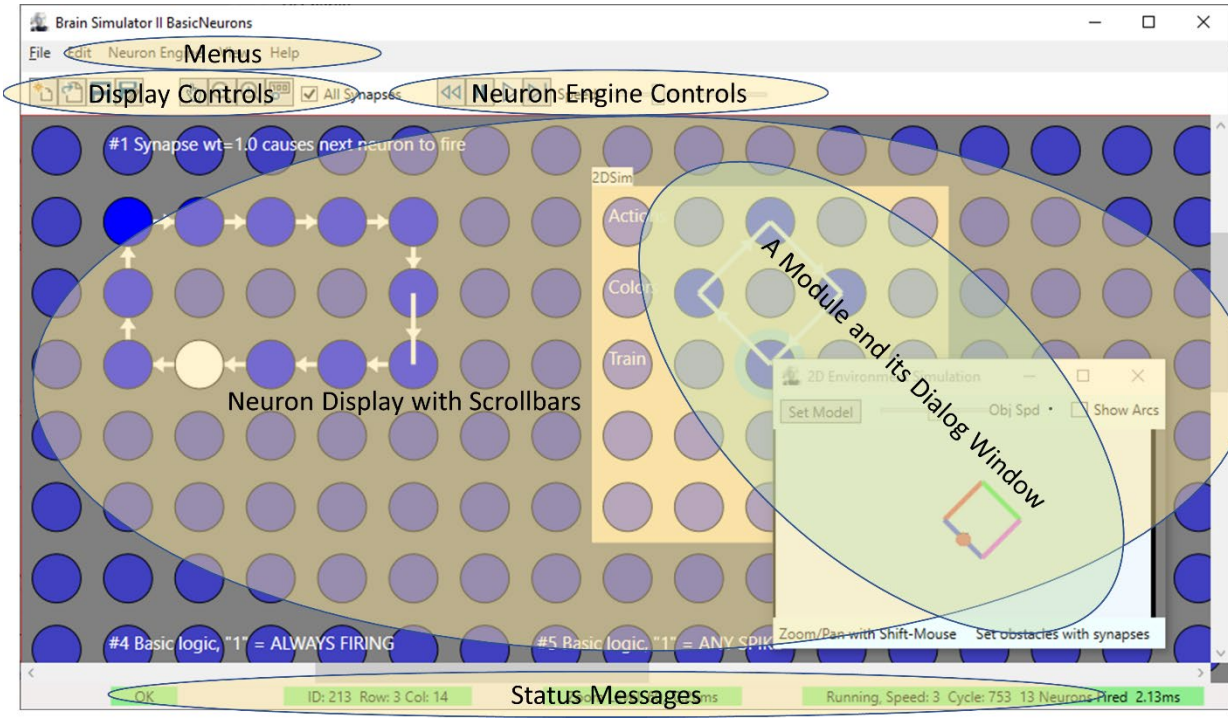
This chapter describes the user interface in detail, but the *Brain Simulator* is a standard GUI program with a few menus, button controls, and a display of the neuron array and its Modules. As such, you may choose to just look at the pictures to get an idea of how the system works and start using the program. If you run into questions, this chapter makes a good reference.

Overall Layout

The majority of the screen is devoted to the display of the array of neurons and synapses—the network display. Colors represent the current membrane potential of the neurons or the weights of synapses. Labels within the neuron array are for reference and are typically not used for computation.

In general, the menus control the network files, Neuron Engine, and the display. The command bar has two sections, one for the network display and control and the other for Neuron Engine control. Status messages are displayed at the bottom of the screen.

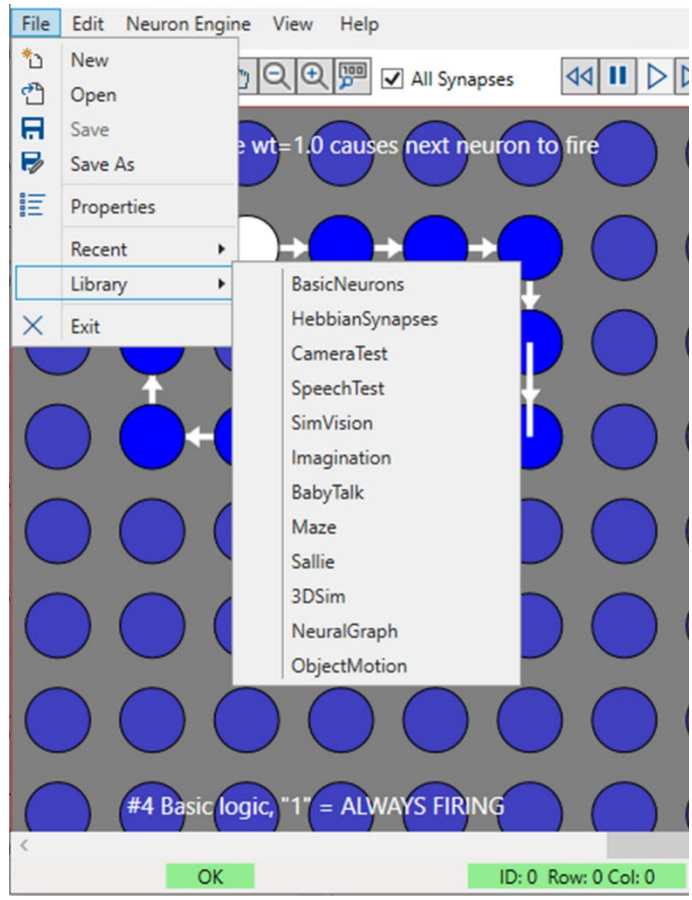
The interface has been tested with a billion neurons, so being able to zoom to the desired location within the neuron array quickly is important.



The overall layout of the Brain Simulator *user interface is focused on the display of the neuron array. Menus let you load and save the neuron array to a file. In the command bar, one cluster controls the display and another controls the Neuron Engine.*

Controlling Network Files

As mentioned previously, networks are stored in files in XML format. You can think of the *Brain Simulator* as an editor for these files. With this thinking, all the file control is similar to that found in a text editor. When the program starts, it will default to displaying the neuron array which was last used. If that file isn't accessible, a sample blank neuron array will be displayed.



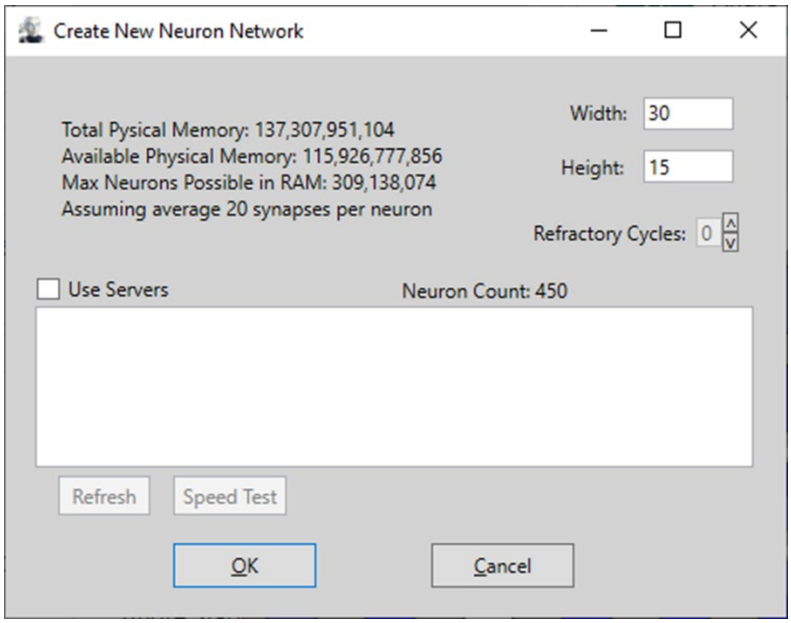
The "File" menu controls saving and restoring neuron networks as is detailed in this section. The "Library" section lists the networks included with the program.

New Files

To create a new file, use the "File | New" command to bring up the file creation dialog. The dialog will display the free memory on your computer and estimate the maximum size of network you can create. Like a word processor, creating a new file does not save it. You must subsequently perform a "File | Save" (or "Save As") if you wish to subsequently retrieve your network.

In the New Neuron Array dialog box, the Rows and Columns numbers you enter define maximums for the network. You can increase these later but you cannot subsequently reduce the size of a network. On the other hand, you *can* create a new one of different

size and use the clipboard functions to copy the network content from the original to this new file.



The dialog for creating new files lets you set the size of the Network and (optionally) create random synapses for every neuron and configure the Network to use multiple Neuron Servers.

You can initialize the refractory period for the network (this can be changed later on in the "Neuron Engine" menu). The refractory period defines the time base for the network and can be left at 0 for most networks. On this dialog, you may also set up the use of Neuron Servers, which is covered later in this chapter.

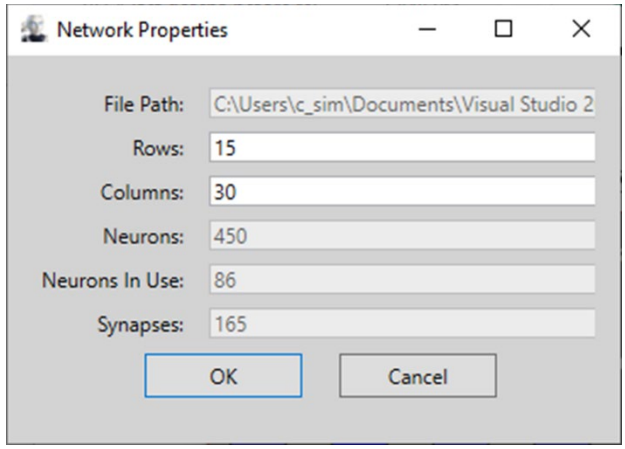
When you press OK, the new network will be created and displayed. With large networks (many millions of neurons), network allocation may take a significant amount of time and a progress bar will show the progress of allocating neurons and random synapses.

Loading Networks

The commands, "File | Open", "File | Recent", and "File | Library" (which lists the network files included with the distribution) all load the related network file. If the network has "Notes", they will be displayed (read-only) when the network opens unless you check the "Don't show this again" checkbox. If you subsequently select "Edit |

Notes", the same notes will be displayed in an editable form and these edited notes will be saved with the Network when it is next saved.

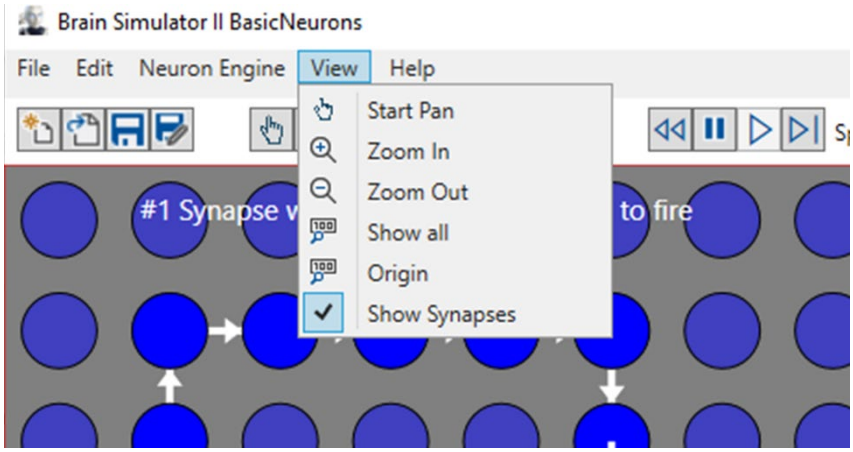
The "File | Properties" command will display a dialog with information about the current network. You can increase the number of rows or columns. Neurons are only "in use" if they are connected by at least one synapse or have a label.



The File | Properties dialog displays basic information about the Network.

When you close the program, you will be prompted to save the network file (except in the case of library networks) on the assumption that since this is a real-time processing program, the content will necessarily have changed. Library network files are installed in a read-only directory. If you would like to modify a library network file, you'll need to use the "Save as" command to save it somewhere else.

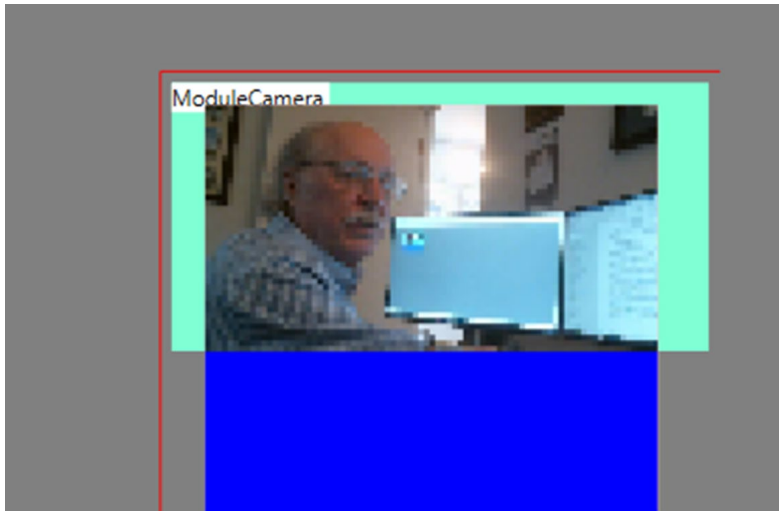
Controlling the Neuron Display



The "View" menu shows the commands for controlling the neuron display.

Neurons and Synapses

Neurons are displayed as disks or as rectangles or individual pixels depending on the display scale. Neurons are usually shown as disks in the figures in this text. As the display is zoomed out to show more neurons, such as for image processing, faster pixel and rectangle displays are used.



Showing the display of the Camera Module when it is zoomed back to show many neurons. Each neuron may be a single pixel on the display. This also shows how, at very small scales, only the neurons within a selection box are displayed (described later).

Color codes for Neurons: Neurons are color-coded to indicate the state of their internal charge (membrane potential). A bright blue neuron has a charge of zero. A dull-blue neuron is not "in use" as it has no label and no synapses connecting to or from it. A firing neuron is white (an internal charge of 1.0 or more). In between, increasing neural charge goes through a rainbow from light blue to green, yellow, orange, and red. Neurons that have the Color model selected will instead display the color of their internal value.

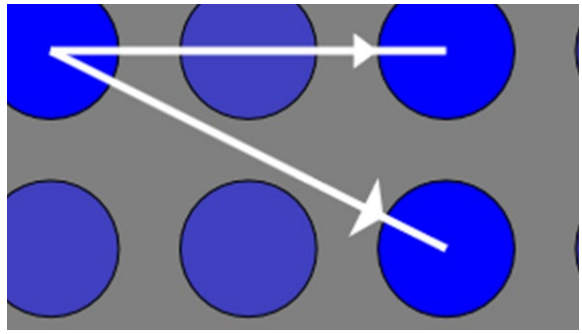
A single neuron may be surrounded by a light-blue circle. It is the "current" neuron and will be the target of paste, move, or multiple-synapse actions which are described later.

Neurons with the LIF model will show "L=" followed by the leakage rate. Neurons with the Burst model will show "B=" followed by the number of spikes in each burst. Neurons with the Random model will show "R=" followed by the mean firing rate (in cycles). Neurons with the Always model will show "A=" followed by the firing rate (in cycles).

Enabling/disabling the Display of All Synapses: The checkbox labeled "All Synapses" and the menu command "View | Show synapses" control whether or not synapses are displayed for the entire network. If the display of synapses is not needed it should be disabled as a large number of synapses can slow display performance. When the display of all synapses is disabled, only synapses from individual neurons which have been selected to "Show Synapses" will show. For slight differentiation, these individually-selected synapses display in front of neurons while others are behind neurons.

Only synapses with either a source or target neuron within the current neuron display will be shown—a synapse connecting two off-screen neurons will not show even if it crosses the display area. For UI performance reasons, a maximum of 2,000 synapses will be displayed; others will be ignored. In this case, there will be a warning in the status bar indicating that there are too many synapses to display.

Synapses are displayed as a color-coded line with an arrowhead indicating the direction of the connection. A narrower arrowhead indicates that the synapse is "Fixed" (see below).



The wider arrow on the lower synapse indicates that its weight may be changed by the Neuron Engine. The weight of the upper synapse is fixed. These synapses are both white, indicating a weight of 1.0.

Color codes for positive synapse weights are the same as neuron colors. Negative (inhibitory) weights progress from light gray to black over the range of 0 to -1.

Mouse Cursor Shapes

As the mouse cursor moves through the neuron display it can take one of several forms depending on its location and the function which will be performed.

Either Shift key is pressed or the Pan Display Control (button with the same icon) has been pressed. Drag to reposition the display.

The mouse cursor is between neurons. Drag to select a group of neurons. Ctrl+drag to append another rectangle to the selection. You can create odd-shaped selections with multiple selection rectangles. Right-click to display the selection Context Menu.

The mouse cursor is over a neuron. Click to fire the neuron and select it as the "current" neuron. Drag to create a synapse. Right-click to show the neuron context menu.

The mouse cursor is over a synapse. Right-click to display the synapse context menu.

The mouse cursor is on a Module. Drag to move the module. Right-click to display the Module context menu (see below). Similar direction arrows which appear when the mouse cursor is near the edge of a Module allow you to resize the Module by dragging the edge.

Display Control

Neuron arrays can be huge and have been tested with up to a billion neurons, so it is valuable to be able to display the area of interest easily. You can zoom and pan through the display several ways, with control buttons, scroll bars, and just the mouse. As the display scale changes, the amount of detail in the display changes to help keep the display-update speed as fast as possible. As you shrink the display, most neurons are not displayed and red lines form a grid with reference numbers which can be useful in locating specific areas of the network if it is large. Areas of neurons can be displayed selectively by adding a selection.

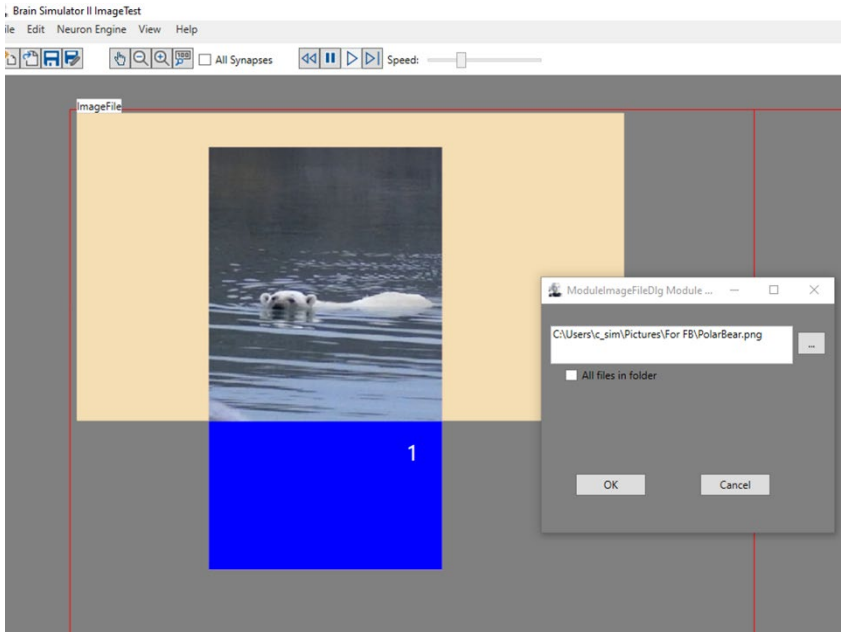
Any time the mouse cursor is in the neuron display, pressing either keyboard Shift key changes the mouse cursor to a hand to allow you to pan the display with the mouse.

The mouse wheel changes the display scale as do “Zoom In” and “Zoom Out” buttons and related menu commands. The “Zoom to Origin” button and related “View | Show All” and “View | Origin” menu commands can get you quickly to a desired display. The “Zoom to Origin” button toggles between “Show All” which shows the entire network” and “Show Origin” which shows the upper left corner of the network. On networks which will fit completely on the screen, these two displays may appear the same.

Like the pan function, the scrollbars at the bottom and right side of the neuron display allow you to reposition it horizontally or vertically. The arrow buttons at the ends of the scrollbars will move the display by one row or column of neurons at a time. The areas of the scrollbars between the thumb-track slider and the arrow buttons will move the display one screen-full at a time when clicked.

The neuron display updates itself independently of the Neuron Engine. This means that every update of the neuron display might represent multiple cycles of the engine. The display represents a

snapshot of the neuron states at a specific point in time as the Neuron Engine is paused momentarily for the display to be updated. The elapsed time used to update the display is shown in milliseconds in the Status Bar and can be useful in learning how various display options change the display rate (all the parameters which control what items are displayed at which zoom levels are easily changed by programmers by editing the DisplayParams.cs source file).



This display with a million neurons is zoomed smaller until only the reference grid and Modules are seen. Each reference grid square is 250x250 or 62,500 neurons. The area of displayed neurons is created by adding a selection to the desired area. This can be useful if the neurons of a particular area represent an image or the overall firing pattern is useful. In this instance, the selection area overlaps ModuleImageFile which can read image data from a file. The surrounding neurons in the selection are bright blue, indicating a membrane potential of 0.0.

Controlling the Neuron Engine

In general, the Neuron Engine runs continuously and there is no need to stop the engine to change the network (just as there is no need to stop your brain when various connections change within it). Internally, some functions (like save) will pause the engine while the

process completes so the neuron state of the entire array is consistent.

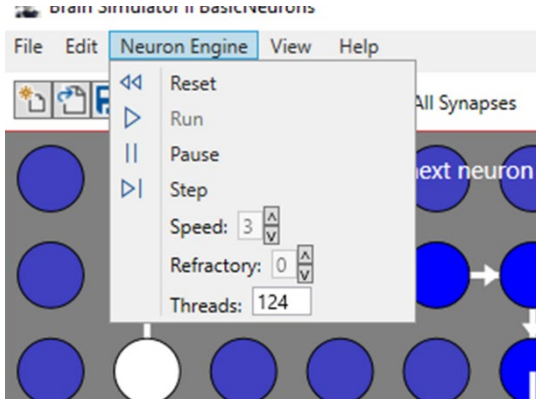
The Neuron Engine speed is controlled internally by adding a delay at the end of each processing cycle. When the speed is set to zero (the slider is all the way to the left) the inter-cycle delay is 1 second. When the speed is set to 10 (the slider is all the way to the right), the delay is 0 so the Neuron Engine is running as fast as it can.

The Neuron Engine status display shows the speed, a cycle counter, the number of neurons which fired in the previous cycle, and how many milliseconds elapsed during an engine cycle, which is a moving average of the previous 100 cycles.

You can control the Neuron Engine either with the Controls or the Neuron Engine menu. “Reset”-ting the engine calls the Initialize method on all the Modules in the network. Run and Pause start and stop the Neuron Engine. “Step” will execute a single cycle of the engine and will also pause it if it is running. “Speed” duplicates the function of the Speed Slider and introduces a delay between engine cycles.

“Refractory” changes the refractory period (in engine cycles) for all the neurons in the network. This should be changed cautiously as virtually all networks rely on a consistent refractory period and changing it will likely require corresponding changes to the network to keep it working.

“Threads” shows or changes the number of computing threads used by the Neuron Engine. This does not normally need to be changed but can be useful for optimizing Neuron Engine speed. Normally, the neurons in the array are distributed equally among the threads. Assuming an even distribution of firing and synapses, the computational load will be distributed evenly among the CPU cores. This doesn’t usually need to be changed except for high-performance testing. Since the UI thread is always running, it’s a good idea to set the thread count to one less than some multiple of the number of cores. If you have four cores and set the thread count to five, the first four threads can run in parallel but the fifth (orphan) thread will have to run when another has completed, potentially doubling the overall time it takes to process a cycle.



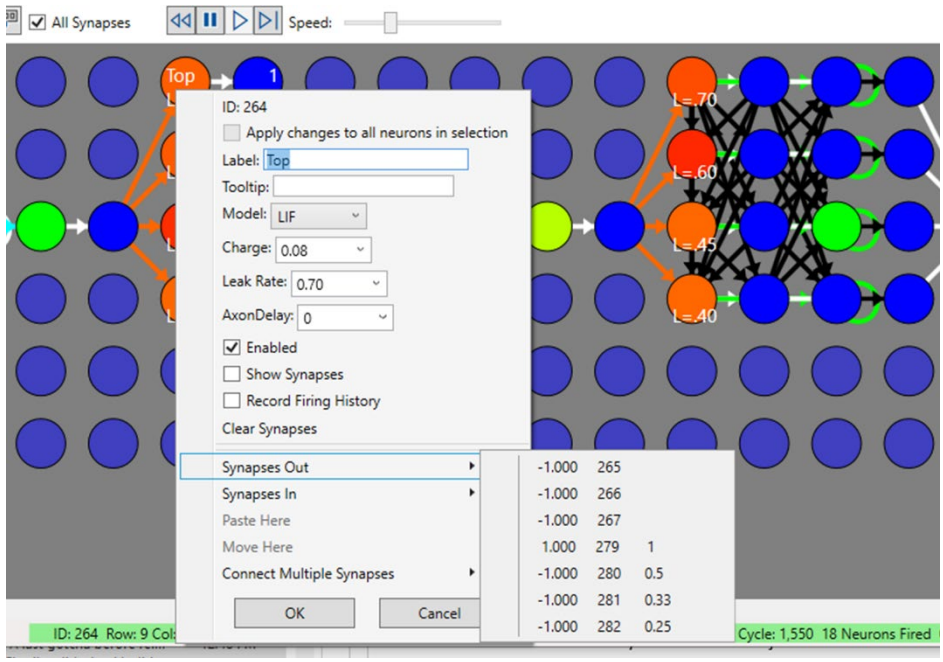
The Neuron Engine menu allows you to set the various engine parameters.

Editing Networks

Neurons

Clicking a neuron will cause it to fire. If it was firing continuously, clicking the neuron will cause it to stop firing. Double-clicking a neuron will disable or enable it.

Right-click a neuron to display its context menu. Each neuron has a numeric ID, which is the location of the neuron within the neuron array.



Right-click any neuron to display its context menu. Sub-menus show incoming and outgoing synapses. In the synapse menu, the first number is the weight, the second is the target neuron ID, and the third (if it exists) is the neuron label. In this case, the neuron labels are numeric too.

The ID is followed by a checkbox which is only available if you have selected one or more rectangular groups of neurons (covered later). When this box is checked, any changes you make to this neuron will be applied to all the neurons in the selection. Only edited entries will be applied so, for example, you'd like to set the neuron charge to 0.08 on all the neurons but the charge on this neurons is already 0.08, you'll need to change it to something else and back. Changed fields which will be applied show a green background.

Each neuron may carry a label. If you add a label, it can be just a few characters and will reside within the neuron disk or it can be longer, extending beyond the disk of the neuron to act as a notation in the network. Neuron labels are not typically used in computation but can be used to reference the neuron. Duplicate labels are allowed but a warning is displayed if you set a label which occurs elsewhere in the network. The neuron label may be used when editing a synapse or when pasting a selection as described later.

If the neuron label is set, you'll also have the option of entering a tooltip. This text will show as your mouse cursor moves over the neuron. It is handy to be able to keep the neuron label to just a few characters and add a longer explanation to the tooltip if needed.

In the dropdown, you can select the model to be used for this neuron. This is followed by the neuron's charge or membrane potential. You can edit it. There is a dropdown but you can key in any value you like. New values you enter will be added to the dropdown for future use.

For this and the following entries, numeric text entries must be syntactically correct. Illegal entries will show a red background and will be ignored. Numbers which are outside the usual range for the value will have a yellow background but will be set as requested. If you set a neuron charge to 1.23, for example, the system will set it, but the neuron won't necessarily make use of the value outside the range of [0,1].

Below this, several model-specific parameters may be displayed which are described at the end of this section. In this case, the Leak Rate and Axon Delay are specific to the LIF model.

The checkbox, "Enabled," can be used to temporarily disable a neuron or group of neurons.

The checkbox, "Show Synapses," will display the synapses originating from this neuron. It does not override the "All Synapses" option which displays synapses regardless of the settings of individual neurons, or various limits in display size and synapse count which may prevent synapses from showing.

The checkbox, "Record Firing History" will begin recording for this neuron and will open the firing history window if it was not already open.

The "Clear Synapses" entry will remove all the incoming and outgoing synapses on this neuron.

Lists of synapses to and from this neuron are also available. Each list shows the weight and the target neuron (or source neuron in the case of an incoming synapse). The neuron's label will be displayed if it has one. If you click in the area of the weight, it will open the synapse context menu so you can change the weight. If you click on the target neuron ID, the context menu for the target (or source)

neuron will be opened. If that neuron is not visible or the new context menu wouldn't fit at the neuron's location, the neuron display will pan so that the target neuron is near the upper left of the screen.

The entry "Paste Here" will insert the content of the clipboard at this current location. The entry "Move Here" will move the content of the current selection to this current location. These are covered in detail under "Clipboard" below.

The "Connect Multiple Synapses" command has a submenu with three commands which act in concert with a selection. The command "Selection to Here" and "Here to Selection" will add a synapse with the current default characteristics between the current neuron and every neuron in the selection. The two commands differ in the direction of the synapses. The command "Mutual Suppression" will add a synapse of fixed weight -1 between every two synapses in the current selection.

For the following commands: If the "Apply changes to selection" checkbox is checked, the change will apply to all the neurons in the selection:

- Changing the neuron's label.
- Setting the neuron's charge.
- Changing the neuron's model.
- Changing any of the parameters custom to the model.
- Selecting whether synapses are displayed.
- Adding or removing the neuron from the Firing History.

If you change the label, the new labels in the selection will be incremented from the label you set.

Model-Specific Entries

For neurons other than the IF and FloatValue models, custom parameters are available as follows:

Color model: the Charge is displayed as the hexadecimal value which is the ARGB representation of the color.

LIF model: the Leak Rate and Axon Delay are available. The Leak Rate is the fraction by which the charge will be reduced in each engine cycle. The Axon Delay is the number of engine cycles after a neuron fires when the synapses will deliver their weights to their target neurons.

Random model: the Mean is the number of engine cycles that the neuron will fire if the Std Dev is zero—the average firing rate. The Std Dev is the standard deviation of a Gaussian distribution of spike times around the mean. Setting the Std Dev to -1 will disable the random firing.

Burst model: The Count is the number of spikes that will be created and the Rate is the number of cycles between spikes.

Always model: The Delay is the number of Neuron Engine cycles between spikes.

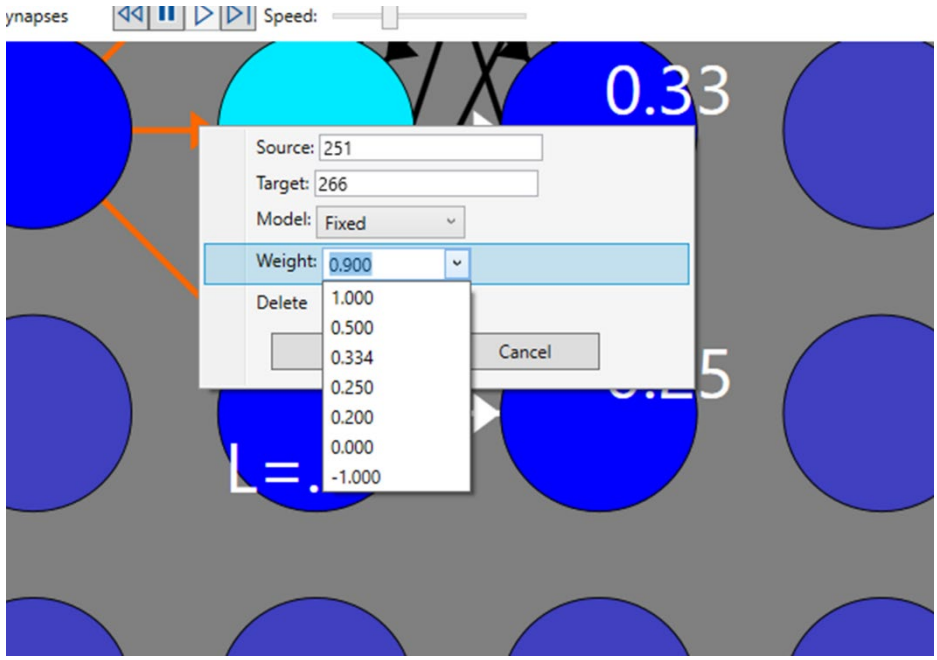
Synapses

To add a synapse, position the mouse cursor over the source neuron (note the up-arrow cursor) and drag to the target neuron. The synapse will be added with default characteristics (initially weight=1.0, Fixed). You can Undo an added synapse with "Edit | Undo" or Ctrl+z.

Right-click a synapse (when the cursor is) to display the synapse context menu and change its characteristics. You can change its weight by selecting a new weight or entering a weight in the text box. You can press DEL on the keyboard or select "Delete" from the menu to delete the synapse.

The source and target neurons will be shown. If they have labels, these will appear, otherwise the neuron IDs will be shown. You can move a synapse by entering a new source or target neuron IDs or labels. In the event that there is more than one neuron in the network with the same label, the one with the lowest ID will be used.

When you display the context menu for any synapse, the characteristics of that synapse are set as the default characteristics that will subsequently be used when new synapses are added. This means that a quick way to add a synapse of a given weight is to right-click on a similar synapse to set the defaults, press ESC to close the menu, then add your new synapse.



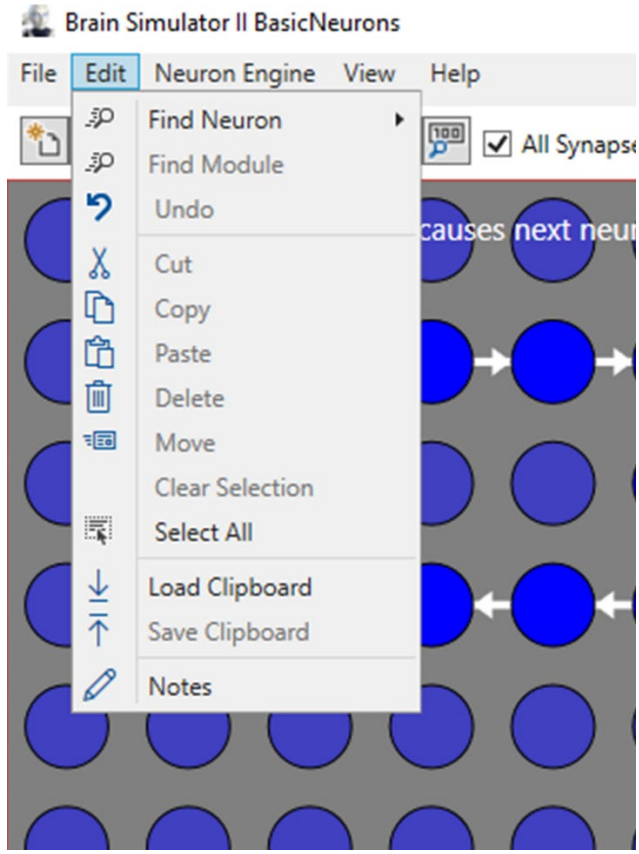
Right-click on a synapse to display the synapse context menu.

The "Model" dropdown selects the synapse model used by the Neuron Engine to alter the weight during execution. The Neuron Engine uses a lookup table to determine how much to change the weight.

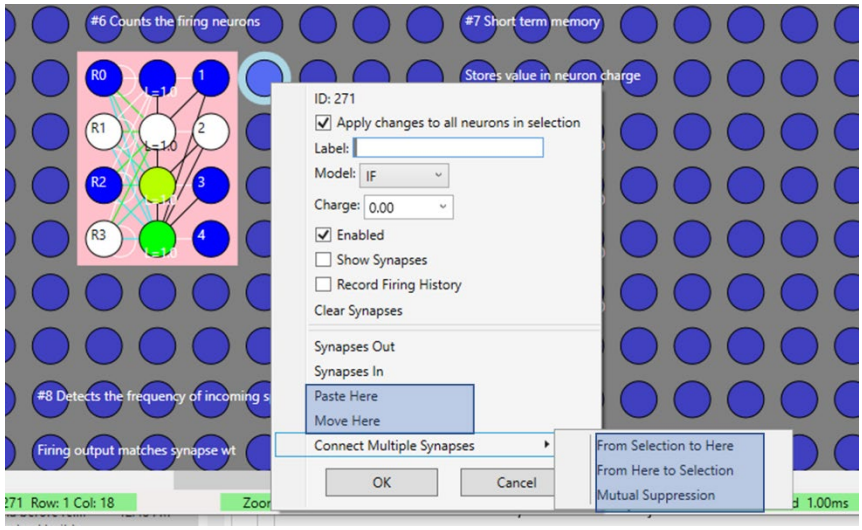
You can set the weight directly by entering it numerically in the text box or by selecting one of the common weights on the dropdown. You can delete the synapses with the "Delete" command or by pressing the DEL key on the keyboard.

Clipboard

The clipboard is a powerful function based on the expectation that the brain or an AGI will consist of repeating patterns of neurons. The clipboard can be used to easily replicate areas of functioning neurons from one network to another.



The "Edit" menu contains commands for accessing the clipboard. For larger networks, the "Find Neuron" and "Find Module" commands can be useful.



Using the clipboard makes it easy to create multiple copies of useful functioning clusters of neurons. The synapse pattern is copied along with the internal state of neurons so the pasted cluster can continue the computation of the cluster it was copied from.

When the mouse cursor is between neuron disks and is displayed as a cross, areas of neurons can be selected by dragging the mouse across the area. The selected rectangle is shown in pink. To make a complex selection shape, you can hold the Ctrl key and select multiple rectangular areas. These rectangles may overlap or be discontinuous but form a single selection.

Once a selection is made, several commands are available. As mentioned in the previous section, certain commands on individual neurons and synapses will be applied to all neurons and synapses in the selection. For example, you can change the model of all the neurons in a selection by changing the model of any neuron in the selection.

Several commands which act on the clipboard are available on the “Edit” menu, on the neuron’s context menu, and via keyboard shortcuts.

Copy: A selected area can be copied to the clipboard. The clipboard can be considered to be a network in its own right. Once copied to the clipboard, this smaller network can be pasted elsewhere into the same network, or a different network can be opened and the clipboard content can be pasted into a second

network. The copy command can be performed with the "Edit | Copy" menu entry or with Ctrl-c. Synapses that cross in or out of the selection ("boundary neurons") are included in the clipboard if the neurons they connect outside the selection have labels.

Delete: Clears all the neurons in the selection and removes any synapses which are sourced by or targeted to neurons in the selection. The Delete command can be performed with the "Edit | Delete" command or with the "Del" key.

Cut: this combines the Copy and Delete commands. The Cut command can be performed with the "Edit | Cut" menu entry or the Ctrl-x key.

Paste: This copies the content of the clipboard into the network at the location you specify. You will be warned if the paste will overwrite neurons in the target and the paste must fit within the bounds of the neuron array. The Paste (and Move below) command needs a destination location within the neuron array. This is provided automatically if the paste command is selected from the neuron context menu but to use the "Edit | Paste" or Ctrl-v command you must first set the target location by clicking a neuron which will be displayed with a light-blue ring. Synapses which cross the clipboard boundary will be replicated/stretched if the neuron outside the boundary has a label.

Move: The move command is slightly different in that it does not require copying to the clipboard. You can simply select a group of neurons and use the command "Edit | Move" or the "Move Here" on the neuron context menu. Unlike copy/paste, all synapses that cross in or out of the selection are stretched whether or not the neurons have labels. You can also drag a selection but if neurons in the selection collide with other neurons in use, you'll be warned. If you proceed, these in-use neurons will become part of the selection and will be dragged along with it.

The content of the clipboard can be considered a network in its own right. The command "Edit | Save Clipboard" will prompt you for a file name and will save the content of the clipboard to a network XML file. That XML file can subsequently be opened and used as a network on its own. The command "Edit | Load Clipboard" will read a file into the clipboard for pasting into another network. Boundary neurons are not saved to a file.

The clipboard is local to the *Brain Simulator* and is not usable for passing data to other applications.

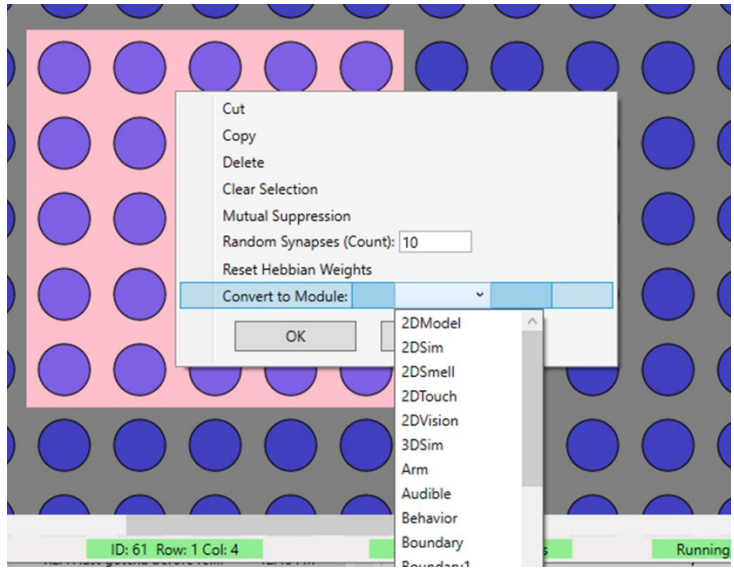
Other Selection Functions

If your cursor is a motion arrow (✥) within a selection, you can right-click for the selection context menu. You can copy, cut, and delete the content of the selection. You can clear the selection.

"Mutual Suppression" will add synapses between all pairs of neuron in the selection with a weight of -1.0.

"Random Synapses" will add synapses with random targets and random weights to all the neurons in the selection. The number of synapses added to each neuron is controlled by the "Count" textbox which should be set first.

The Command "Reset Hebbian Weights" will set the weights to zero of all non-fixed-weight synapses which are sourced in the selection area. This can be useful for testing networks which store information in Hebbian weights.

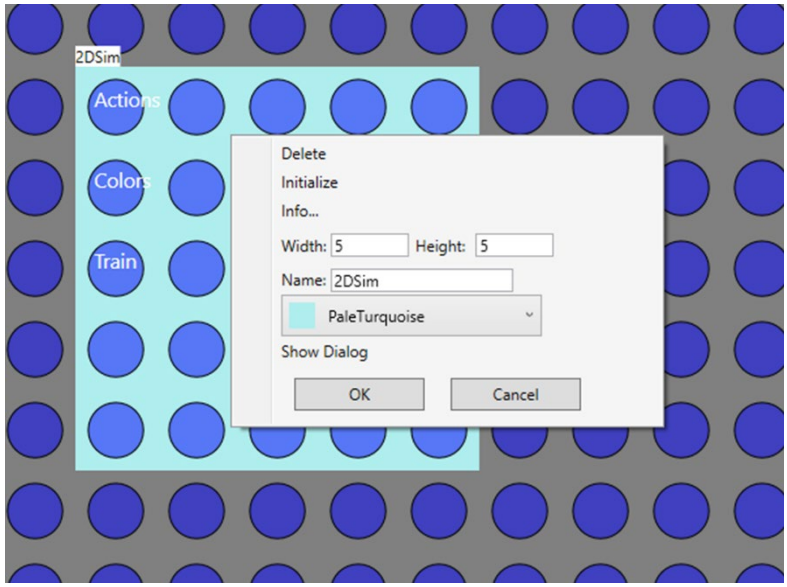


The context menu for a selection has a dropdown that allows converting a rectangular selection area into a Module.

Lastly, a single selection rectangle can be converted to a Module by selecting the Module type from the dropdown list. The function

of the module is defined by the software within the Module itself (See Chapter 6 for a list of Modules).

Once a Module has been added to a network, the mouse cursor changes ins the area within the Module (but outside neuron disks). The cursor changes into appropriate drag handles which can be used to move or change the size of the Module.



The context menu for a Module.

You can right-click a Module to bring up its context menu. From the context menu, the Module can be deleted, initialized, or named. If the Module has an internal description it can be displayed with "Info…".

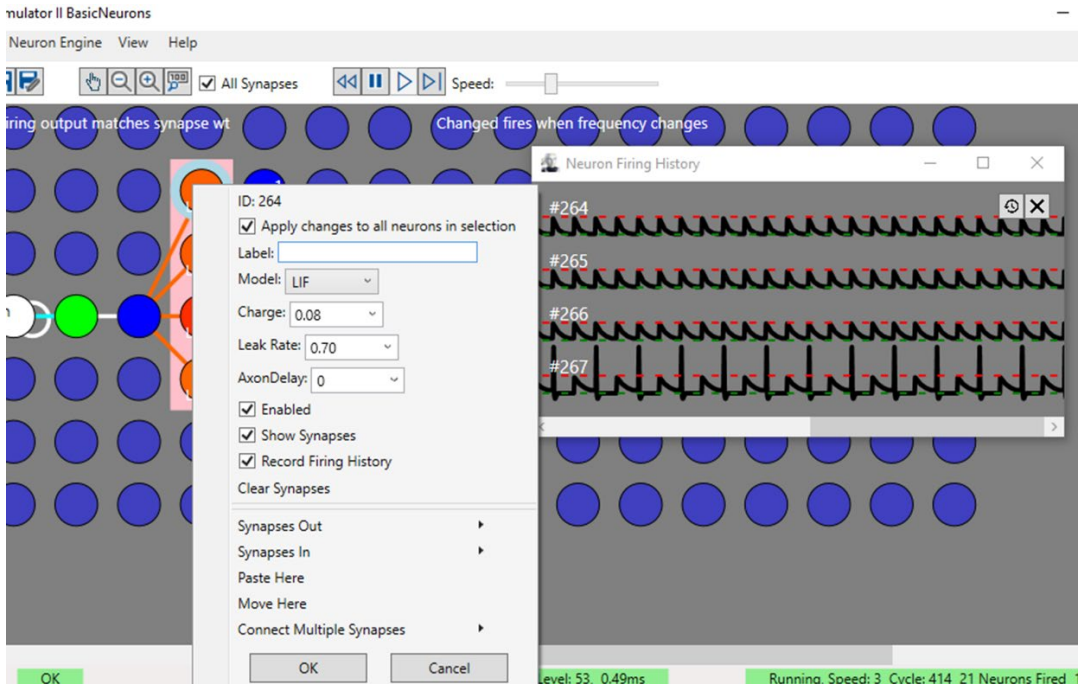
The dimensions of the module can be modified. This is the best way to create large Modules with specific dimensions which would be difficult to set with the drag handles. Use caution because there is no check to prevent Modules from overlapping each other which could lead to unpredictable results.

If the Module has a dialog box, you can display it from the context menu. Each dialog is custom to the Module.

Firing History

Firing history can be collected on any set of neurons. To add neurons to the firing history, check the "Record Firing History" checkbox in a

neuron's context menu. History will commence when one of the recording neurons fires. Then firing history will be collected continuously as long as the engine is running.



The Neuron Firing History window shows the value of the internal charge for the selected neurons in the form of a timing diagram. Labels at the left of traces show the neuron's label, if it has one, or the neuron's ID number.

To control the Firing History display window, you can use the buttons in the upper right to either remove all the previous firing history but continue recording or to clear all the firing and stop recording.

If the mouse cursor is within the Firing History window (with a normal arrow cursor), the mouse wheel can be used to expand the display. Once expanded, the scrollbar at the bottom of the window can be used to scroll through the content in the window, even when expanded data is still being collected. When the scrollbar is at the right-most edge of the window, live data is displayed as it is collected.

The traces in the window are labeled with the neuron ID or the neuron label if it has one. The order of the traces in the window can be controlled by dragging the labels to new locations.

The charge level of each neuron is recorded and displayed accurately but the waveform of the spikes themselves are synthesized.

Multiple Servers

The Neuron Server can be operated simultaneously on any number of computers on a local network. The IP addresses of the machines must all be in the same family (the first three fields of the IP4 addresses must be the same). One machine must operate the *Brain Simulator II* program to control the other servers and the control machine may also be a server. Servers may have different performance and resource characteristics and this should be taken into account when configuring the system.

Because of its use of the network, your computer's firewall will need to be edited to allow this network traffic. If you are using Windows Firewall, there is an application being developed which will automatically make these changes. If you are editing your own, you need to know that there are two applications which need to be allowed: BrainSimulator.exe and NeuronServer.exe. Both applications need to be able to send via UDP, both broadcasts and targeted messages. Neuron Server uses ports 49001, 49002, and 49003 while Brain Simulator uses ports 49002 and 49003.

There are two steps needed to use the Neuron Server. On each server machine, start the server program (NeuronServer.exe). On the *Brain Simulator* machine, use the "File | New" dialog to configure which neurons in the network will reside on which servers. Check the "Use Servers" checkbox and press "Refresh" to find all the Neuron Servers on the network. The servers will be listed in the text window and the neurons will be initially assigned to them by distributing them evenly. You can edit the textbox to assign the neurons differently if you choose.

As of the current release, server configuration information is not saved with the network and you can only use the New Network dialog box to set up server configurations. To load an existing network to multiple servers: Create a new with the correct neuron counts for the network to be loaded. Then load the network and it will use the configuration you juse entered.

Performance capabilities and limitations: The Neuron Engine in the Neuron Server is identical to the one used in the *Brain Simulator*'s single-computer configuration. This means that the performance of the Neuron Engine will be identical to the non-server version for synapses that do not cross a machine boundary (and all neurons). Synapses that *do* cross a machine boundary will be substantially slower. Also, the overhead of multiple machine control adds a few milliseconds to each cycle, and the performance of the user interface is substantially reduced. See Chapter 12 on "Performance" for details.

Keyboard Shortcut Summary

Ctrl-z: Undo

Ctrl-a: Select all

Ctrl-x: Cut, copy to clipboard and delete from neuron array

Ctrl-c: Copy to clipboard

Ctrl-v: Paste from clipboard

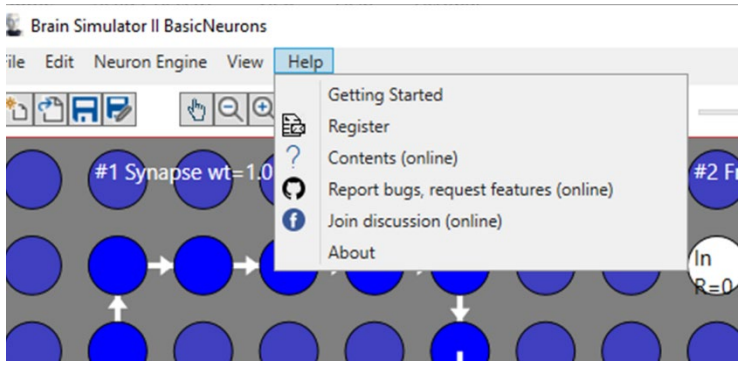
Ctrl-m: Move neurons in selection area to target neuron

DEL: Delete neurons/synapses in selection

ESC: Close Selection or close context menu without saving

F1: Help Getting Started

Help and Support



The Help menu shows many ways you can get support or contribute to the project.

Getting Started: Opens a browser window and displays the overview help file with suggestions for first-time users.

Register: Opens a browser window to the software registration page.

Contents (online): Opens a browser window with the online help content which includes this chapter.

Report bugs, request features (online): Opens a browser to the GitHub repository for the project. After you've created a GitHub user name, you can enter bugs or suggest features directly to the "Issues" database. You can download the source code too. If you want to contribute the project, request access to the code base.

Join Discussion (online): Opens a browser window to the Facebook BrainSim group page. Join the group for all the latest information.

About: Displays version and contributor information.

Video Links

"Brain Simulator II Overview"

http://futureai.guru/videos?id=141

Chapter 8:
The Programming Interface

This chapter gives an overview of the programming interface for the Brain Simulator II. This is a more strategic overview and not a description of features and interfaces as these are covered in the code itself, which can be downloaded from GitHub at the URL: https://github.com/FutureAIGuru/BrainSimII. Even if you're not a programmer, you might want to read this chapter to get an idea of the capabilities and limitations of the Brain Simulator and AGI development in general.

There are three ways to program the *Brain Simulator II* in addition to creating networks through the user interface:

- by interacting directly with the Neuron Engine, bypassing the user interface.
- by writing new custom Modules.
- by making any other customizations, such as adding new neuron models.

To modify the source code, you'll want to download the free Visual Studio (Community Edition) from Microsoft. In setup, you'll need to enable the C# and C++ languages and any other language you choose to use.

The Neuron Engine interface

The Neuron Engine is written in C++, has been carefully optimized, and is extremely fast. There is a C++ demonstration program (CppEngineTest) included in the source code which shows how the functionality of the engine can be accessed.

There is a similar program in C# (CsEngineTest) that uses an intermediate translation library, the NeuronEngineWrapper. With this second method, the Neuron Engine is accessible from any language supported by .NET. The *Brain Simulator* itself interfaces to the Neuron Engine through a translation module

(NeuronHandler.cs) which should also be useful for a programmer using the Neuron Engine from a managed .NET language.

This second method uses the same underlying engine code so there is no loss of performance on the neuron array itself. But because of the language translation, the interface to the engine is somewhat slower. This may be an issue if a network is to be defined with many millions or billions of neurons or synapses.

This programming level does not support Modules. The major steps in using either the C++ or C# (.NET) interface are:

1. Define the neuron array and give its dimension. At this programming level, the neuron array is one-dimensional—the two-dimensionality is all created by the *Brain Simulator*'s user interface.
2. Set neuron parameters and add synapses.
3. Repeatedly call the "Fire" method. The engine will execute a single engine cycle every time this is called. After each call to "Fire" you can retrieve the count of neurons that fired.
4. Retrieve neuron information from the engine to get the results of the computation.

Within the Neuron Engine, the principal objects are NeuronBase, SynapseBase, and NeuronArrayBase. Every NeuronBase object contains a Vector of SynapsBase objects and the NeuronArrayBase contains a vector of NeuronBase objects. You can have multiple NeuronArrayBase objects if desired. Only physical memory limits the number of neurons and synapses you can use.

Adding a New Neuron or Synapse Model

Within the Neuron Engine, edit the NeuronBase.h file and add your new model name to the enum near the beginning of the file. Then, edit the NeuronBase.cpp file and edit the "Fire1" and "Fire2" methods to create the functionality for your new model. Within the code, you can see how the various existing models are handled. This is all that's needed to make the new neuron model work in the Neuron Engine.

To make your new model also accessible from the *Brain Simulator* user interface, you'll need to edit Neuron.cs and add your new model type to the modelType enum and a tooltip to the

modelToolTip array. Then you'll need to edit NeuronView.cs, edit the GetNeuronView method to change the way the neuron displays (if you want), and likewise the SetCustomCMItems method if you want the context menu to display custom parameters.

The process for synapses is similar except that the enum change is in the file SynapseBase.h while the functionality of the synapse is in NeuronBase.cpp. For the user interface, similar changes will be needed for Synapse.cs and SynapseView.cs.

The Module Interface

As previously described, *Brain Simulator* Modules are an extremely powerful programming tool.

To create your own Modules, there are template files (under the "Tools" folder in the source code). These can be added to Visual Studio to make creating Modules and their dialogs easy. Then within Visual Studio, you can "Add new item" and just as you might select C# Class, you can select "Module" to create a new Module or "Module Dlg" to create a custom dialog box for a Module.

A Module can do anything the computer can do without restriction. All Modules are inherited from the class ModuleBase. Module Dialogs inherit from ModuleBaseDlg. These take care of all the housekeeping.

The objects within the Neuron Engine are mirrored in the Brain Simulator User Interface code on as as-needed basis by the objects Neuron, Synapse, and NeuronArray. As such, within a module, you have full access to controlling the engine. Be aware that when you request neuron information from the NeuronArray, a request is forwarded to the Neuron Engine and the returned data is properly reformatted. If this results in performance issues, you may choose to use the Neuron Engine interface directly as described above.

The Module contains two principal functions:

1. The Initialize method is called when the Module is added to a Network, whenever the "Initialize" command is selected from the context menu, or the Neuron Engine is initialized. Note that the Initialize method is *not* called when the network is loaded from a network XML file as

this would change the state of the network which is otherwise unaltered by Saving and Opening.

2. The Module's "Fire" method is called once prior to each cycle of the Neuron Engine.

If you need to reinitialize something whenever a file is loaded, the method SetupAfterLoad is provided. An example of this is initializing the underlying speech engine any time a file using the SpeechIn Module is loaded. Similarly, if your module uses data structures that don't stream well into XML, the Method SetupBeforeSave can be used. An example of this is the removal of potential circular links in the Universal Knowledge Store Module. These are mirrored by corresponding changes in SetupAfterLoad that restore the content to its original state.

All public properties are automatically saved and restored to the Network file. If, for some reason, you need a public property that you don't want to store, precede its declaration with [XmlIgnore].

Lastly, there is a vast array of services that are useful for manipulating neurons and accessing other Modules. Principal among these are GetNeuron and GetNeuronAt, which can return a neuron object for any neuron in the network. Given the neuron, you can query or set its properties, add or delete synapses, etc.

At the time of this writing, Modules are executed sequentially and the order is defined by the placement of the Module's upper left corner in the neuron array, top-to-bottom, then left-to-right (in numerical order of ID). In general, this makes little difference because a module which doesn't process on one cycle will be able to on the next.

Modules may expose public methods which are accessible to other modules. Generally, Modules should communicate by setting neuron values rather than by method calls because it makes the Modules more generally useful. However, there are instances where the use of neurons would be tedious and direct method calls are more convenient. An example of this is described in the Universal Knowledge Store chapter.

Are you Cheating? The Limits of Plausibility

Philosophically, intelligence might take many forms and there may be intelligences that work differently from human intelligence. At this point, however, human intelligence is the only general intelligence we know about, which is why the *Brain Simulator* uses it as a model.

On the other hand, there are good reasons to ignore biological plausibility on occasion, but when you do, it's useful to know how far you've strayed from the "true path" and why. As an example, we know that your brain is capable of estimating distance given the differences in images received by your two eyes. We could speculate on how this might be accomplished in neurons *or* we can write a few lines of trigonometry code to accomplish the same task. This is done in the Module2DVision file even though it is implausible that any portion of your brain works with the use of floating-point trigonometric functions.

It's fine to develop AGI in any form that works. At the same time, it's a good idea to notice how your AGI differs from human AGI so you can highlight areas of additional AI risk which should be addressed.

Chapter 9:
The BasicNeurons Network

This chapter is a sample of the "Notes" included with every library network. Notes can contain any desired text but typically include:

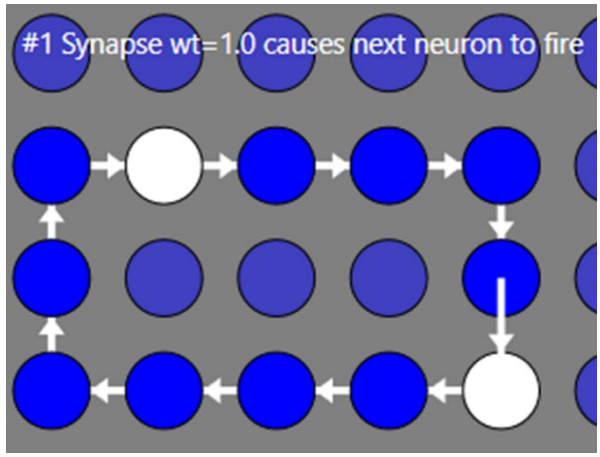
- Purpose—what's the point of this network.
- Things to Try—ideas about things you might learn from this network.
- Current State of Development—known bugs and suggested future capabilities.

*When any Network is opened, a "Network Notes" section is displayed if it has one. It includes sections on the **Purpose** of the Network, **Things to Try**, and the **Current State of Development** of the Network. When you build your own Networks, you can add and edit notes as well.*

Purpose:

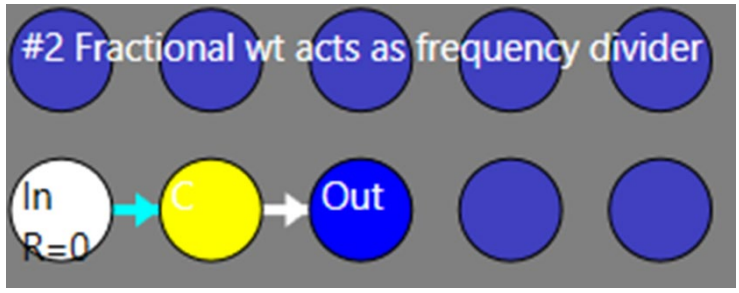
This network illustrates some capabilities of the basic Integrate and Fire (IF) and Leaky Integrate and Fire (LIF) neuron models.

The neurons in cluster #1 are connected, one-to-the-next, by synapses of weight 1.0, so the firing of one causes firing of the next.



By connecting one neuron to the next with a synapse of weight 1.0, each firing neuron will cause the next to fire in sequence and firing neurons will chase each other around the loop. The center-right neuron has a synapse in front of it because it has "Show Synapses" selected.

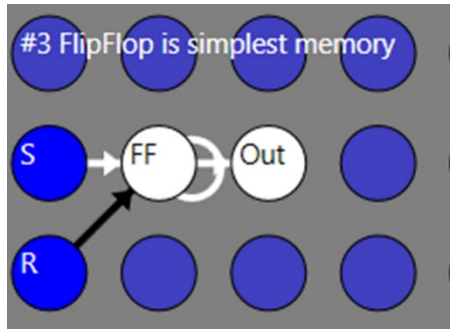
Cluster #2: the left-most is set to always fire. It is also connected to the center neuron with a synapse of weight 0.25. You can see the color change as the center neuron accumulates charge. When it reaches its threshold, it will fire and cause the right-most neuron to fire once every fourth cycle.



This tiny circuit shows how neurons spontaneously act as frequency dividers. Of the three neurons, the left-most fires on every cycle while the center fires at a rate dependent on the weight of the incoming synapse.

Below that in cluster #3 is the simplest possible memory circuit. Neuron FF is either firing all the time or it is not and so represents a single bit of storage. The neuron is connected to itself with a synapse of weight 1.0 so whenever it fire it will contribute enough charge to itself so it will fire again. The R (Reset) neuron is connected to FF

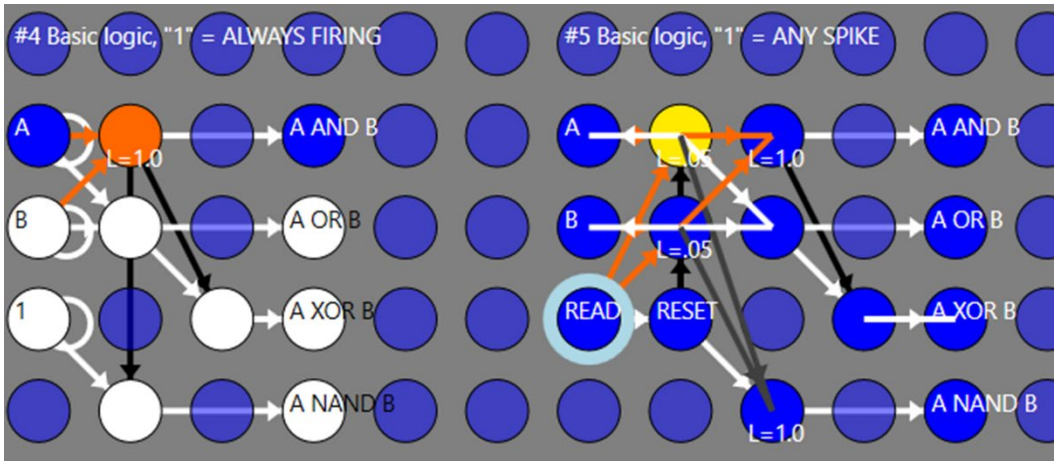
with a synapse weight of -1.0 so firing R will cause FF to stop firing. S (Set) is connected with a synapse of weight 1.0 so firing it will cause FF to start firing continuously. This works because the axon delay is equal to the refractory period; both are 0 in this case. If the refractory period were longer, the signal out from FF would arrive during its refractory period and would be ignored so this circuit would be a bit more complex.



The simplest bit of memory is also called a Set-Reset Flipflop. It has two states, in this case, firing or not firing, and so can store a single bit of information.

Cluster #4 shows basic digital logic functions of AND, OR, XOR, and NAND implemented in neurons using a logic model of always-firing is "1" and never firing is "0". This is an important demonstration because it proves that neurons form a functionally complete set and so could be used to implement ANY logic circuit. A CPU could be constructed from neurons just as a brain could be constructed from transistors.

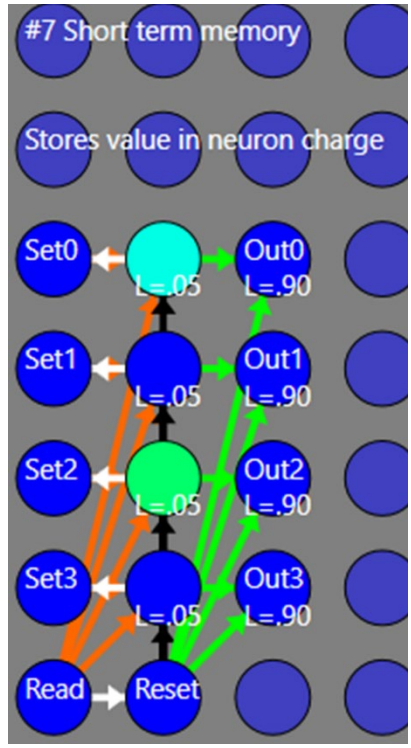
To the right in Cluster #5 is a similar set of logic circuits using a logic family where "1" is represented by ANY SPIKE and "0" is represented by no spike at all. This uses much less energy because continuous firing is not needed but outputs are only valid after the READ neuron spikes.



These two circuits show how digital logic can be implemented in neurons. In the left-most set, the basic logic elements are created with a logic 1 defined as firing on every cycle. A more plausible system on the right only requires neuron firing when the logic values change—it consumes much less energy for similar performance.

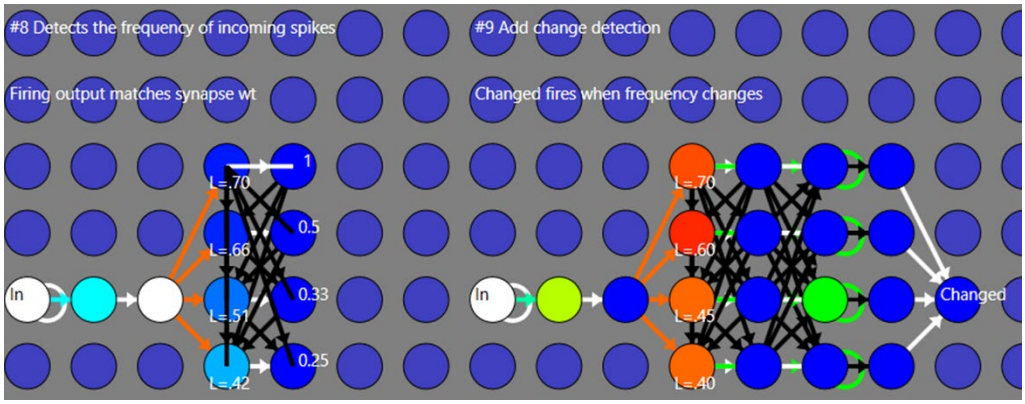
Cluster #6 should a simple circuit to count the number of firing neurons in a group. The right-hand column indicates the number of neurons (R0-R3) which are firing.

Cluster #7 shows another mechanism which can be used for short-term memory. In this case, the neuron's internal charge can also store a bit of memory. In the center column of neurons, if the charge is greater than 0.1, this represents a 1. If the charge is lower, it represents a 0. The output is only valid after the Read neuron fires. This memory is fast but fades with the leakage of the neurons and so must be refreshed. If the memory is not read, it will be lost. Other memory mechanisms are described in the HebbianSynapses network.



The Neurons, Set0-3 are the data inputs to this four-bit memory. Each contributes a 0.9 charge to the neurons in the center column which is not enough to cause spiking. When the Read neuron spikes, it contributes 0.9 and any neuron with a bit stored in it will fire the corresponding Out neuron.

Clusters #8 and #9 are two circuits for detecting the firing rate of the input. In each group, the left two neurons are connected by a synapse which generates a defined spiking rate. Cluster #8 detects the frequency while cluster #9 adds the ability so that a single spike is output when a frequency is detected but there is no spike output otherwise. The multiple single-spike outputs are OR'ed together in the "Changed" output. It will spike whenever the input frequency changes.



These two circuits convert from a rate-based encoded signal to parallel signals. The right-hand version includes memory so it can detect if the incoming firing frequency has changed.

Things to Try:

In cluster #2, right click the synapses from In to C and change its weight. Note how you can change the firing frequency of Out.

In cluster #3, click the "S" and "R" neurons to change the firing state of the FF flip-flop. This is a fast memory mechanism with this type of neuron model as it stores a bit in a single neuron. This is also a good opportunity to demonstrate the history window by selecting all three neurons, right-clicking, and selecting the “Record firing history” checkbox.

In Cluster #4, click the "A" and "B" neurons to exercise various digital logic functions [This will make sense to people with EE and CS experience]. A neuron with a synapse to itself will fire continuously after it is clicked and will stop firing if clicked again. Note that a "1" neuron which always fires is necessary to create an inverter (for NAND).

Repeat with cluster #5, the "1"=ANY SPIKE model. Once you click "A" and/or "B", you need to click "READ" for the logic to perform.

In cluster #6, select different combinations of "R" neurons and notice that the appropriate output neuron always indicates the number of neurons which are firing. The right-most column eliminates noise in the output with additional suppressing synapses but is not strictly necessary.

To use the "Short-Term Memory" circuit in cluster #7, click one or more "SET" neurons to store information. Click "Read" and observe that the memory content is set to the "Out" neurons. To store a new value, click "Clear" to clear all memory cells. Because the bits are stored in the internal charge of the LIF neurons, it decays over time and needs to be refreshed periodically with a "READ"--just like DRAM.

You can build your own network:

Right-click any neuron or synapse to see its state and edit it. You can use the checkbox to add more neurons to the firing history window. In that window, you can drag waveform labels up and down in the history window to reorder the waveforms. You can change the model used to calculate a neuron's function.

You can drag the mouse cursor from one neuron to another to create a new synapse. Then right-click the synapse to set its weight. New synapses will default to the characteristics of the most-recently-selected synapse.

You can zoom and pan the neuron display by holding the Shift key and using the mouse wheel or dragging the display. You can also use the scrollbars or the buttons below the main menu bar. Notice that this network is 30x15 or 450 neurons but the simulator works with millions.

When the mouse cursor is between neurons it changes to a cross and you can then drag to select a group of neurons. You can then move to a clear neuron area, right-click a neuron and Move Here to move the neurons. Other standard clipboard commands also work. If you change the model of a neuron within a selected area, all neurons in the selection can be changed. Holding the Ctl key while selecting lets you create a selection with multiple rectangles.

Current State of Development:

This network represents the basic neuron models, engine operation, and user interface and is reasonably robust. Please report any bugs you encounter.

Chapter 10
The HebbianSynapses Network

Purpose:

This Network demonstrates the use of three different synapse models with spike-timing-dependent plasticity which is referred to generically as Hebbian learning. This involves synapses whose weight can be changed by the firing timing of the neurons they connect. The change in synapse weights is generally accepted as the principal mechanism which underlies learning.

There are a number of parameters which can contribute to synapses plasticity:

- The current weight of the synapse.
- The relative spike-timing of the neurons it connects.
- The range of weights the synapse can take on, for example [0,1] or [-1,1].
- The rate of variation of the synapse weight.
- Whether the variation is linear or follows some other algorithm.

The algorithm for weight variation could be any combination of these factors. In biological synapses the specific algorithm is not known and the experimental variation is large. Accordingly, the *Brain Simulator II* supports any number of experimental models for synapse variation. The ones currently implemented are:

- Fixed—the weight cannot be modified by the Neuron Engine.
- Binary—the weight is either 0 or 1 and changes in a single engine cycle.

- Hebbian1—the weight range is [0,1] and varies via a lookup table so that intermediate weights are stable when connecting two neurons. For example, a synapse with weight 0.25 will cause the target neuron to fire every fourth cycle so the amount of increase added to the weight when the target fires is 4 times the amount of decrease.
- Hebbian2—the weight range is [-1,1] and the weight varies so that weights will effect pattern recognition. When there are four incoming synapses, the maximum weight is .25 so that all 4 must fire to cause the target to fire. The weight follows a tanh function so that it is most stable at its maximum and minimum values.

The Complexity of Synapse Plasticity:

One of the points of this network is to show the complexity of building a system with variable synapses. In the BasicNeurons network, all the synapse weights were fixed and the function of the various clusters was completely predictable. In the cluster in this network, if there were no fixed-weight synapses, the clusters would quickly degrade and not function.

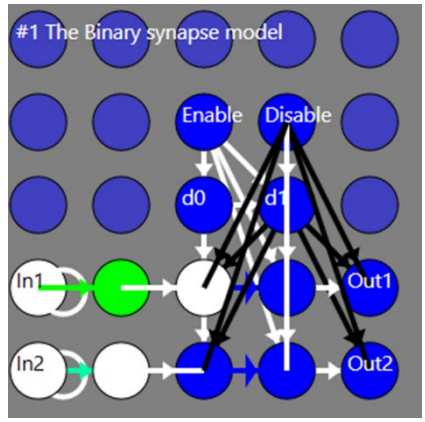
It's easy to say that "Neurons which fire together, wire together." But when thinking about biologically plausible neuron and synapse models, this is easier said than done. The binary synapse model is the simplest possible implementation and in cluster #1, it takes multiple control neurons and fixed synapses to control the weights of a few synapses. With the more complex models, it is obvious that it is impossible to precisely control synapse weights. Further, if you could set them to precise weights, they would not maintain those weights and there is no way (plausibly) to determine with the weights are.

That said, variable synapses can be an extremely powerful tool and clusters #3 and #4 show how they can be harnessed to recognize patterns and store structured knowledge.

Things to try:

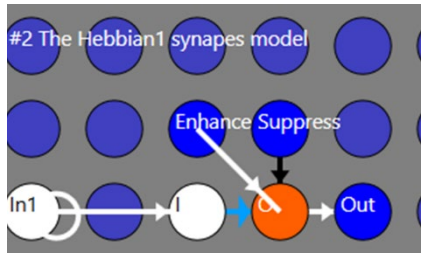
There are four demonstrations in this Network demonstrating the three variable synapse models.

In cluster #1, single binary synapses connect inputs to outputs. By clicking "Enable," the synapse weight is set to 1 and the signal passes through. By clicking "Disable," the synapse weight is set to 0 and the signals are blocked. This shows some of the difficulties of using variable synapses. To set the synapse weight to 1, both the input and output neurons must fire. To set the weight to 0, just the output must fire. In principle, the Enable neuron fires both neurons simultaneously, the Disable neuron fires just the target while inhibiting the source. D0 and D1 are needed for timing. The Out1 and Out2 are used to suppress spurious output spikes which would otherwise occur with Enable and Disable actions because both require firing the target neuron in order to function.



This memory stores the value as the weight of a synapse which is either 1 or zero. The variable synapses are noticeable in the bottom center because their arrowheads are wider. When the weight is 1, the signals from In1 and In2 are transmitted to the outputs, when the synapse weights are zero, the signals are blocked. Four neurons are needed to control the timing to set the weights of the two synapses.

Cluster #2 demonstrates a single Hebbian1-model neuron. Because its weight varies between 0 and 1, it will act as a frequency divider depending on its current weight. By pressing and holding "Enhance", the target neuron will be fired and the weight will slowly increase. The Suppress neuron prevents the target neuron from firing and thus the weight will decrease. If you show the recorded firing history of the I, O, Enhance, and Suppress neurons, you can see how the frequency of the output spiking can be controlled by the Enhance and Suppress neurons.



Again, the variable synapse is in the bottom center. In this case, the Enhance and Suppress neurons force or block the firing of the O neuron and will increase or decrease the weight of the synapse. In this model, the weight will remain near whatever value it is set to.

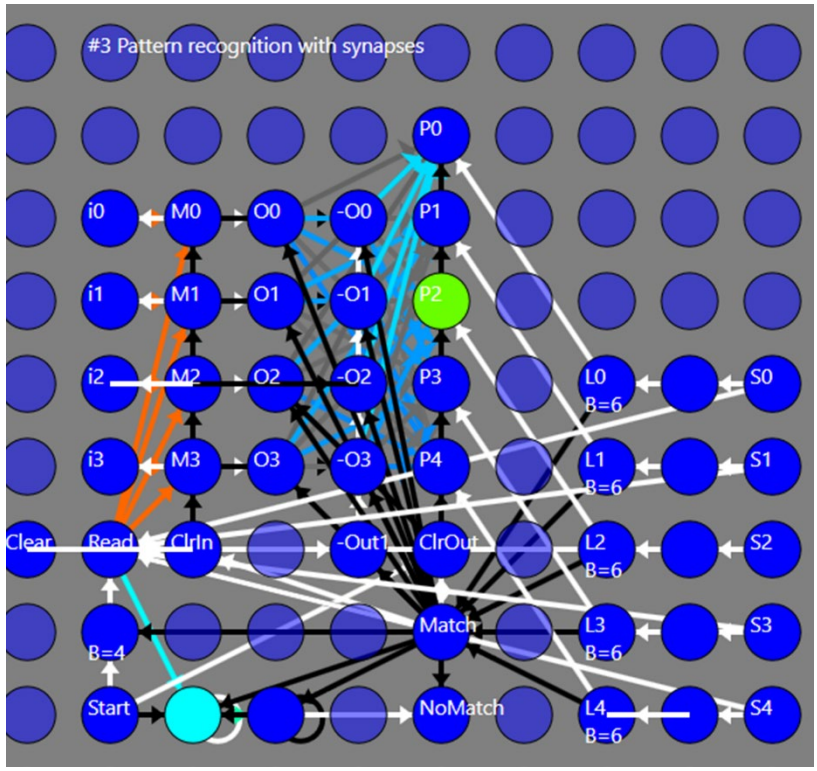
Cluster #3 is a circuit which can recognize patterns of four input neurons. All of the "meat" of this network is in the diagonal synapses between the O, -O and P columns. The inputs (i0-i3) are latched by M0-M3 using the short-term memory circuit from the BasicNeurons network. This way, you can click them in any order and the input timing is not critical.

When you press "Start," the pattern is spiked on O0-O3 and its inverse is spiked on -O0- -O4. By using inverses, this circuit provides equal value to single-bit errors whether it is a 0 which should be a 1 or a 1 which should be a zero. The pattern is presented to the array of Hebbian2 synapses and the output pattern which most closely matched the stored pattern will fire first.

The patterns which are stored in the network as saved are: P0:0000 P1:1111 P2:1100 P3:0011 P4:1010. When a match is found, the "Match" neuron will fire and cause the cycle to stop. If no match is found "No Match" will fire. The closer the input pattern is to exactly matching the stored pattern, the faster it will be recognized.

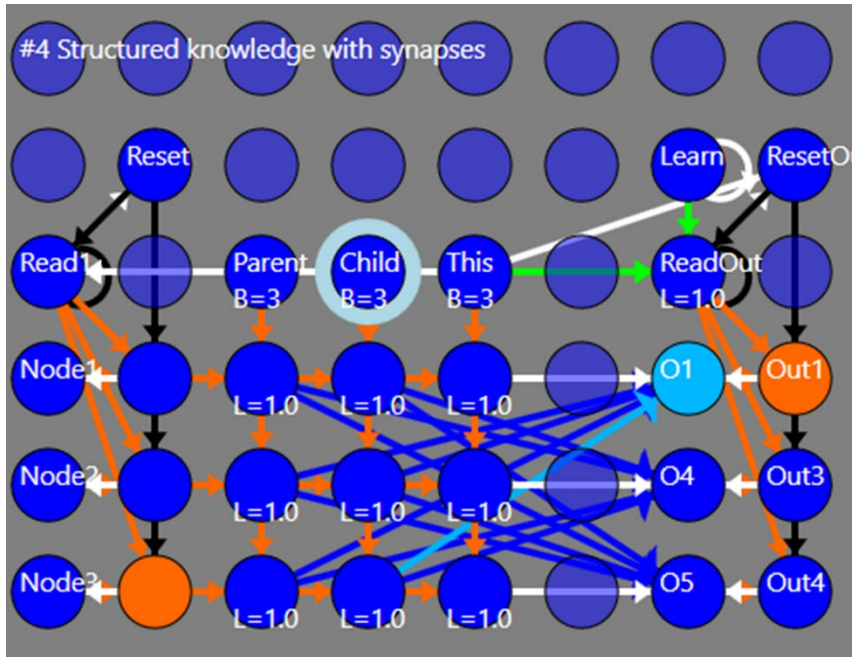
To store new information, start by clearing all the stored information by selecting all the O neurons, right-clicking, and clicking "Reset Hebbian Weights." You can store new patterns by firing "Clear", then inputting your pattern, and then firing one of S0-S4 to store your pattern in the synapses targeting P0-P4 respectively. This will cause repeated firings of the P neuron just after the O neurons and will cause the synapses to adjust their weights to match the pattern. After a pattern is learned, you can repeat the original process of clicking a pattern into the i0-i3 neurons, pressing Start, and seeing the pattern recognized (or not) on a P0-4 neuron. If the data is not cleared before storing a new

pattern, it may take many store cycles in order to store a new pattern. Each of the "L" neurons fires a burst which adjusts the synapses.



This looks a lot more complex than it is. The four inputs, i0-3 represent a pattern which is recognized by one of the neurons P0-4. The L and S columns fire bursts which set the variable synapse weights in the center.

Cluster #4 shows how structured knowledge can be stored in plastic synapses, This closely follows the explanation given in Chapter 11 with the simplest example of a three node graph. The example is to consider that if you know that "red is a color" and "blue is a color" then you can use the structure to answer the "what are colors" and the "red and blue" as an answer.



This circuit shows a neural mechanism by which knowledge can be stored in neurons. A more advanced version of this circuit is shown in the NeuralGraph Network and is then expanded in the Universal Knowledge Store. Information is stored in the diagonal Hebbian synapse weights in the bottom center.

In the cluster, there are three nodes and the relationships Parent, Child, and This. Once data is loaded, if you fire Node1 followed by Parent, the output will fire the parent nodes of Node1. Likewise for Child. The process of getting information into the network requires a few steps. As above, start by resetting the Hebbian synapse weights. To say that Mode1 is a parent of Node2:

- First, fire Node1.
- Fire This to transfer it to the desired output.
- Fire Reset then Node2 to set the new input.
- Fire the Learn neuron to put the network in a learning mode.
- Finally, repeatedly fire Parent to strengthen the appropriate Hebbian synapse.

Current state of development:

This is early-stage development and many other capabilities are being experimented with. Of particular interest is how the synapse weights might adjust to compensate for errors and which pattern neuron should be selected to store a new pattern. Methods have been developed which detect a no-match pattern and store it in the least-recently-accessed pattern. As an alternative, it can also be stored in the least-often-used pattern. Further, there are experiments underway which reset Hebbian2 synapses toward a zero weight if the target fires but no input has fired for a long time. In this way, memories will gradually weaken over time.

Chapter 11: The Universal Knowledge Store

This chapter describes how knowledge might be represented in neurons and then in what I have named the Universal Knowledge Store (UKS). It starts with the development of knowledge in neurons and synapses so you can get an idea of the complexity needed to store the variety of knowledge all of us encounter. Because of this complexity, the development approach was migrated from neurons to Modules for the UKS's higher-level language approach.

After describing the capabilities of the UKS, two applications demonstrate how these can be used. One demonstrates how Sallie can correlate words she hears with objects she sees to learn how words can describe objects. The other demonstrates how traversing a simple maze is representative of a huge area of reinforcement learning where, given some previous experience, you can choose a course of action in any similar situation.

More importantly, this chapter shows how the *Brain Simulator* can merge the abilities of a neural simulator with the power of the computer. Obviously, all the functionality of your brain is embedded in neurons and synapses but the computer, with its different architecture and its own strengths, can be harnessed to great advantage in the creation of Artificial General Intelligence.

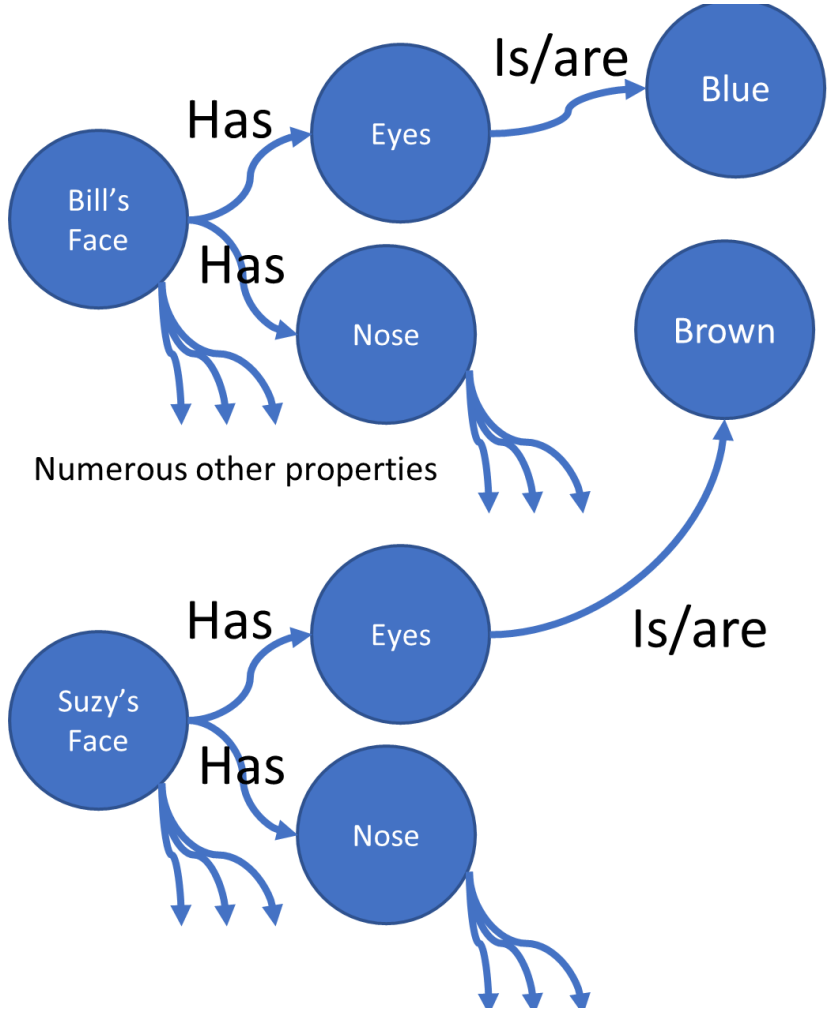
A Brief Introduction to Knowledge in Neurons

All knowledge can be represented. This is a separate concept from the ideas that knowledge can be learned and can be useful, which I'll touch on later. The idea being that representing knowledge is a target while learning is a process toward reaching that target. As of *Brain Simulator* v1.0, the development of the knowledge representation has progressed further than the learning process.

The UKS is an approach for Knowledge Representation which represents a different approach from many neural network proponents who create useful networks without knowing the internal structure of the information. The advantage of the UKS approach is that once you know how some kinds of knowledge are represented in the brain, you can generalize the solution to *any* kind of knowledge.

The Information of Knowledge

Let's start with the idea of recognizing a face—or at least representing the information needed to recognize a face. Every face is recognizable because it has properties, like having eyes and a nose, and each of those properties can have properties, like eyes can be blue or brown (or some other color). The nose might have properties relating to size and shape.



This diagram represents the most basic ideas of how you might represent the information needed to recognize a face.

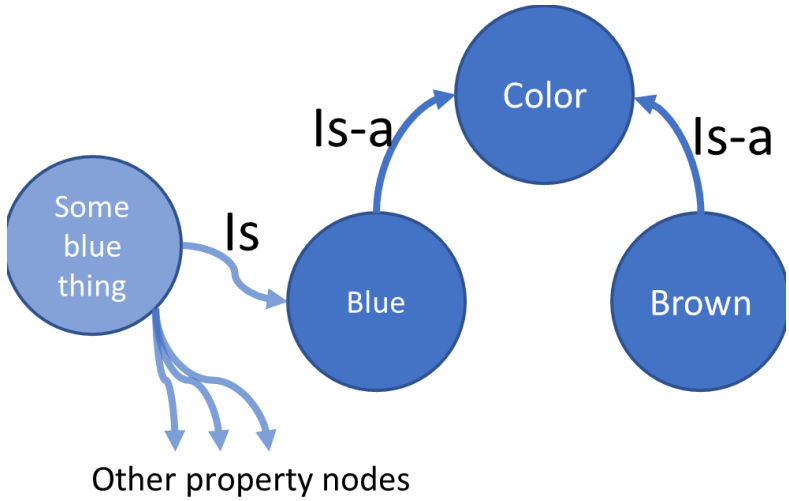
From the diagram, you can know that Bill has blue eyes and Suzy has brown eyes. With considerable extension, you might store enough information to differentiate any human face from any other. You might jump to the conclusion that the circles represent neurons while the arrows represent synapses—but it's too soon to do that as I'll explain in a moment.

The facial recognition experts tell us that about 50 properties of different values are sufficient to uniquely define a face and that an average person can recognize 5,000 different faces. You can see that

any real-world situation will be represented by a graph far too big to show in a diagram. So we simply accept that if we can use computers (or neurons) to implement a structure for knowledge representation, we can implement a larger structure with any number of faces and their characteristics, limited only by processing speed and memory size.

Let's make our knowledge a bit more abstract with a basic example. Consider that all you know is "Blue is a color" and "Brown is a color". Now, you can answer the questions: "What is blue?" (a color), "What is brown?" (a color), and "What are colors?" (blue and brown).

Simple, right? Well, to do this in neurons is not so simple and I'll build up a network that does just this. Let's generalize the question just a bit by recognizing that "blue", "brown", and "color" have a meaning to *you* but at some level are just words or ideas with relationships to one another. Within your brain, these are just spiking neurons with some sort of synaptic connections. In mathematics, you might call this a "graph" which is a collection of "nodes" connected by "edges". To represent this simple knowledge, you might create a graph with three nodes and some edges linking them which might look like this.



Illustrating how certain types of knowledge can be represented in a "graph" of "nodes" connected by "edges."

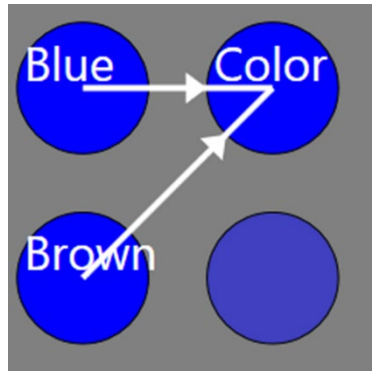
You might say that Color is a parent node of Brown and Blue and that Brown and Blue are Child nodes of Color. An entire field of

"Knowledge Representation" has grown around how you might represent any kind of knowledge and various forms of graphs. A node might have any number of edges, so blue objects like Bill's eyes can be represented by nodes with edges linking to the blue node.

To see this working on the real-world problem of face recognition, here's how you might store the information for a face. You might have a parent node of Face with children Face1 (Bill's face) and Face2 (Suzy's). Every Face has a nose, so you have a Nose parent with noses 1-to-n. Each of the noses has edges to properties like Big or Little or Wide or Thin (which could be children of a more general Size node). Now, I can ask, "What size is Bill's nose?" and access the properties Little and Thin, assuming that there are other property nodes that reference Bill as PersonX which references Face1 *and* the name, Bill.

Implementing a Graph in Neurons

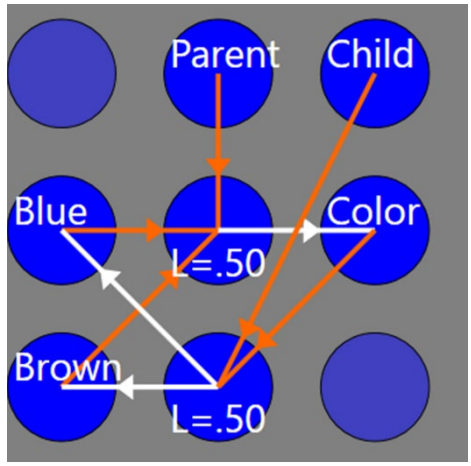
To implement the simple three-node color graph in neurons, let's start by assigning neurons to represent each of the three nodes. To answer the first two questions, you need a synapse that connects Blue to Color and another which connects Brown to Color. Now if the Blue neuron fires (because perhaps you saw something blue or heard the word "blue"), the Color neuron will subsequently fire and you'll know that Blue is a Color. Likewise, for Brown.



If the nodes, "Blue," "Brown," and "Color" were just single neurons, firing either Blue or Brown will cause Color to fire but there is no way to connect Color so it fires Blue and Brown.

Now to answer the third question. With just a synapse from Blue to Color, there is no way to fire Color and get Blue to fire. If you were to add synapses from Color to both Blue and Brown, the trouble

begins. If Color fires, it causes both Blue and Brown to fire which, in turn, cause Color to fire again which causes… so you end up with a situation that for any input, all neurons fire indefinitely. So, we need to add a neuron so that Color will only fire if you want to know the parent of Blue or Brown and another which will cause Blue and Brown to fire only if you've asked for the Children of Color. More generally, by more neurons to each node, we can solve this problem.



By adding neurons to every node, you can build a complete structure so that each node can have parents and children. The center neurons will only fire if two or more input neurons fire. Now, if Blue AND Parent fire, Color will fire. If Brown AND Parent fire, Color will fire. If Color AND Child fire, both Blue and Brown will fire. Imagining that Color also has a Parent and Blue and Brown also have Children begins to show the complexity of solving the most basic knowledge problems in neurons.

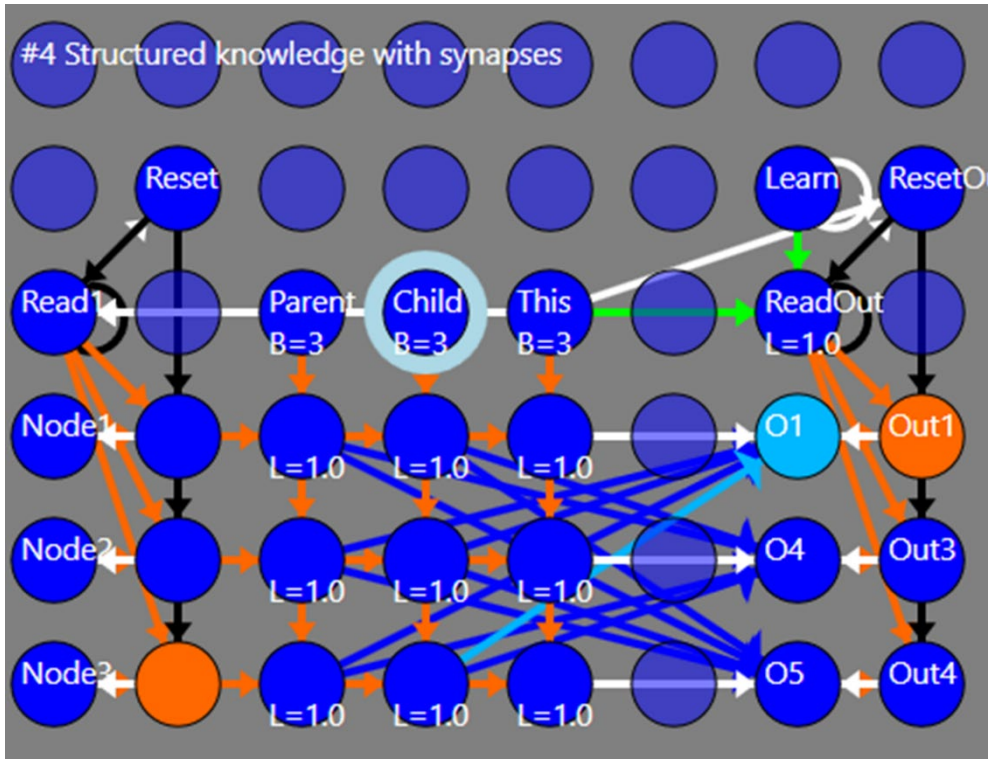
To extend the problem just a little, we have a neuron that fires when you see blue or hear the word, but it must be separate from the neuron which fires for you to *say* the word blue, otherwise, every time you saw blue (or heard the word), you'd also say it. So each node also needs more neurons which determine if you're receiving input or creating output for that particular node. If every node can have a parent and children, and an input and output, each node requires four neurons and you can see the complexity building up.

Then you need another set of neurons that transfer the input to the output so that if you *do* see blue, you *can* say, "Blue" if you want to. I call it the "This" relationship because it answers the question, "What is this?" Likewise, we need a relationship that transfers the output to the input. I call this "Recursion" (labeled "Recur") because

it allows you to ask, “What is the parent of this node?” followed by “What is the parent of the parent?” You ask the first question, then fire the Recur neuron to transfer the output of the first question to be the input of the second, and then ask the same question again.

In the simulator, to expand the capabilities, I added short-term memory to the input and output neurons (so each is actually two neurons). That way, the relative timing of various inputs or outputs is not critical.

Now, we have a basic structure with three nodes represented by eight neurons each. Each node is represented by a number of neurons so it can selectively have a number of different relationships. In this case, the four relationships are Parent, Child, This, and Recur. All the actual information is stored in the weights of Hebbian synapses. Because any node might potentially relate to any other, there must initially be a huge number of synapses even though only a few may be used.



Here's a three-node graph as described if Node1 is Color, Node2 is Blue, and Node3 is Brown. The control neurons are across the top and any number of neuron rows (additional nodes) could be added. The actual data of the structure is the diagonal Hebbian synapses between the nodes. In this case, Node1 is a parent of Nodes 2 and 3 and, conversely, Node1 has the children, Nodes 2 and 3.

Let's assume that the control signals originate somewhere outside the structure (perhaps the hippocampus) and focus on what this network does. If you fire the Node1 neuron, the fact that it fired is stored in short-term memory (below Reset). To get the children of Node1, fire Child—and *voila!* The child node(s) of Node1 will fire on the outputs. Likewise, you can fire Parent to get the parents of Node1 but there are none.

Note that, as described, the network doesn't automatically add a reverse relationship. When we add that Color is a parent of Blue, the connection that lets Color's children include Blue must be done in a separate operation—recall I haven't focused on the learning process in this discussion.

Any number of additional relationships can be added in the form of the parent and child relationship; each one just takes an additional column in the structure. As I built applications, I found two more that were necessary which I named "References" and "ReferencedBy" which can be used to represent any other kind of properties that a node might possess. In the same way that a child relationship is the reverse of a parent relationship, References and ReferencedBy are also inverses.

With this structure and enough neurons and synapses, you *could* represent all the information of 5,000 faces, each with 50 unique properties. Because of the inverse relationships, you not only can ask for a description of Bill's face, but you could also ask, "Who has blue eyes AND a narrow nose?"

Sequential Information

So far, I have considered only objects which have "simultaneous" properties. When you see a face, all its attributes like eye color and nose shape are accessible simultaneously and it doesn't matter if you consider nose size and shape before or after eye size or color. But now consider language. Language requires sequential information because the specific order of phonemes or syllables defines words and the specific order of words defines meaning. If we think of a word as having properties that define how it is pronounced, these attributes must also be "ordered"—it *does* matter that one syllable comes before or after the other.

To represent this, at the very least, each node needs a relationship to indicate which node comes next in the sequence. In neurons, this requires at least yet another relationship column. The next node after node1 is node2.

As a programmer, I would immediately assume that because ordered information requires a "Next" relationship, it implies a back-reference to the previous item in the sequence, a "Previous" relationship, in the same way that Parent implies Child. But you need only consider how difficult it is to recite your phone number backwards, or the alphabet, or any sentence, to convince yourself that your brain only stores forward sequential references (next-node) but not backward references (previous-node). On the other hand, if you hear, "…had a little lamb," you know it's Mary, so there must also be a relationship connecting the multiple nodes of a

sequence back to the first node. Your brain could use this reference to get to the beginning of the sequence and then process it forward to any desired point.

So in development, I added a "Next" relationship and a "First" relationship, which are used to allow any node in the graph to be part of a sequence.

Biological Plausibility

With this structure (and three nodes), I can represent the information that Blue and Brown are Colors and I can list all the colors the structure contains. By extension, I can add rows of neurons to encompass any number of nodes.

Is this biologically plausible? Yes and no.

Because you can answer simple questions about colors, a parent-child structure with relationships *must* exist. Because you can remember for the long term, these relationships *must* be stored in synapse weights. Because you can remember sequential information, some sort of structure *must* exist for that too. But we don't find orderly physical structures like these in the brain so we must assume that the neurons which perform these functions are interspersed with neurons doing other things as well.

That your brain implements these structures *exactly* as I've described is unlikely for a number of reasons. Here are some important ones:

- Redundancy—as I've described the implementation, the failure of any single neuron or synapse can cause a loss of memory. Instead, nodes in the brain likely consist of perhaps a hundred neurons with redundant connections.
- Physical structure—the structure is very orderly and precise and no equivalent has been discovered in the brain.
- Multiple graphs—I've described a single graph but it is likely that various graphs in the brain contain different kinds of information—like visual and audible—and we do know that language and visual processing occur in different areas of the brain.
- Graph Size—the control neurons must connect to every node in the graph, further limiting graph size.

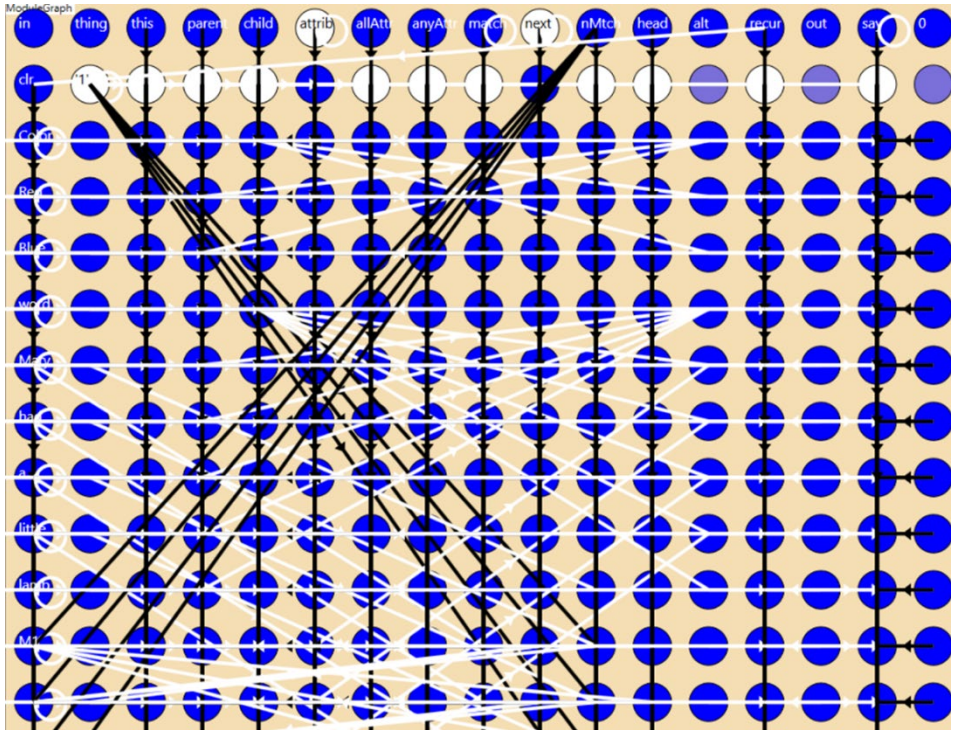
- Storage and retrieval—Although possible with additional neurons, I focused on how information is represented, not how it is stored or retrieved, a key component of the brain.

The objection of, "This is just too complex to be plausible," is not valid. Consider that a horse can walk, see, and avoid obstacles within hours of birth. The neural complexity of those functions (which must be "preprogrammed" by DNA) makes this graph idea seem simple in comparison. The key is that with a node structure consisting of a number of neurons and synapses, the structure can be encoded in our DNA and then repeated many millions of times as the brain develops.

I hope you can see that although the basic idea is simple, a knowledge graph can be built of nodes and edges (or neurons and synapses). With sufficient nodes, a graph can represent information of immense complexity.

The NeuralGraph

Whether the specific structures I've described and implemented in the Brain Simulator exist in your brain is not known but obviously, some equivalent structure *must* exist because you can answer the types of questions I gave in the examples. Within the BasicNeurons network, there is a small demonstration graph that was built by hand.



The NeuralGraph Network implements the functionality described and requires 16 neurons per node. It includes not only Parent, Child, and Reference (called "Attrib" here) relationships but the ability to search sequences of words so you can input "Mary had" and get back the firing sequence "a little lamb." The "Recur" relationship (recursion) allows the system to take the output of one search and use it as the input for another. The Module puts a label on each node so you can see what the content is.

The next step in development was the creation of the Module, ModuleGraph, which is included in the NeuralGraph sample network. It includes a method, AddNode, which can add a new node to the graph and add all the synapses to represent information within the graph. It also includes demonstration methods that allow searching by spoken input and creating spoken responses.

As implemented in the *Brain Simulator*, each node requires 16 neurons. Although it is no longer supported, the NeuralGraph can provide insight into the plausibility of complex graph structures in neurons.

Enter the Universal Knowledge Store (UKS)

The system described above was implemented with a Module ("ModuleGraph") that can create as many nodes as desired and automatically arrange all the neurons and synapses needed to represent it. Within the *Brain Simulator*'s UI, you can watch individual neurons fire as information is stored in, or retrieved from, the structure. This is great for a few dozen nodes but becomes unwieldy for larger graphs. So the next development replaced the neuron/synapse computation with a structured program within a Module to make it easier to experiment with various ideas and structures.

Here were my objectives for the Universal Knowledge Store:

1. Biological plausibility because it is generally equivalent to the NeuralGraph, implemented in neurons.
2. To be able to store any kind of knowledge and relationships.
3. Require little pre-programming…with both the structure and content being learned.

Biological plausibility is not an absolute necessity for intelligence but, as the human brain is the only working intelligence we know of, it seems like a logical place to start.

The Universal Knowledge Store implements a knowledge graph of unlimited potential and complexity. It represents information as a collection of nodes connected by edges. The Module contains only two useful object types, a "Thing" and a "Link" which are concrete implementations of a theoretical node and edge.

The Thing represents anything (a word, a physical object, a color, an action, etc.), and a "Link" connects one Thing to another Thing. While I like to think of a Thing as being analogous to a neuron and a Link as analogous to a synapse, this is a very loose analogy, as I've shown that a single Thing might require a hundred neurons in your brain.

The Link

First, because it's simpler, we'll start with the definition of a Link. We can say that a Link is "owned" by a Thing and targets another Thing in the same way we might say that a Synapse is owned by one neuron and connects to another. The other end of the Link is the Target Thing.

You can see that a Link is also analogous to a synapse in that it connects in a single direction and has a weight or strength. A Link may, likewise, include a weight that can represent the confidence or importance of the Link. The weight of a Link cannot represent any information such as the intensity of a color, as has been explained previously.

In practice, most Links don't require a weight at all. To represent that "Blue is a color", or any other known fact, the weight would always be 1. Blue is either a color or it's not (or perhaps you're not sure, yet). There's a bit more to a link which I'll cover later when I describe learning.

As a software structure, a Link is just a reference (C#) or a pointer (C++) to a Thing along with a floating-point weight value.

The Thing

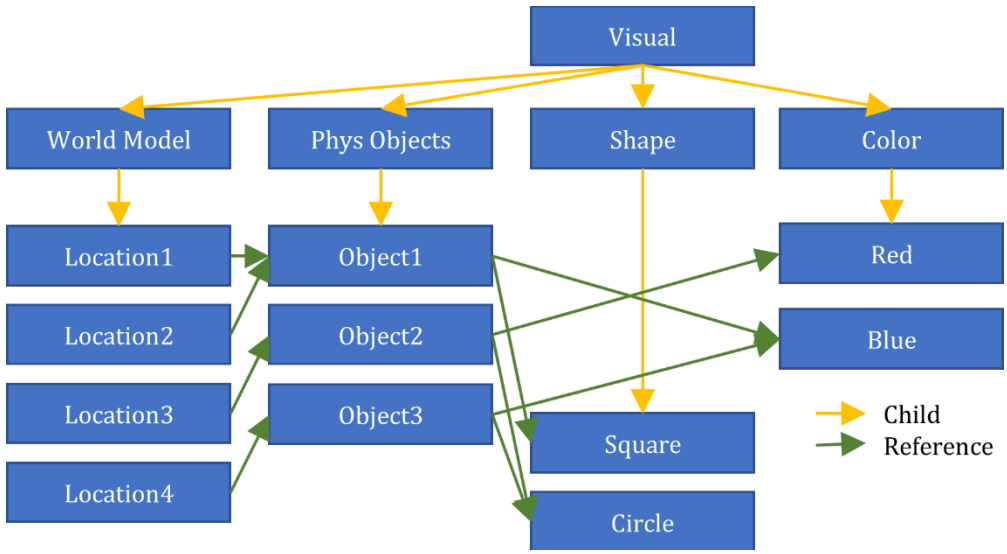
Now for the Thing. First off, each Thing has lists of Links to other Things. In theory, a single list would suffice but for programming convenience, the Thing has specific lists for "Parents", "Children", "References", and "ReferencedBy" Links (more about these in a moment).

In theory, a Thing *has no content*. A Thing's meaning is inferred entirely from the other Things it links to. If you have a Thing with links to "red" and "square", you know it represents a physical object. If it also links to a Thing representing a position, you know it represents a *specific* physical object. As you'll see, I've taken liberties with this constraint as well.

For convenience in creating a Tree-like structure of Things, there is the ability to create parent-child relationships as described earlier. Each Thing has a list of parents and a list of children. By using a List structure, each Thing can have any number of parents and any number of children. This way, we can represent unlimited "is-a" relationships.

The software method which adds a parent Link to a Thing, also automatically adds the child Link to the parent Thing. That way if you tell the knowledge store that "Red is a color", which adds color as a parent of red, the UKS is immediately able to answer the "What are some colors?" question and get red among the results.

A portion of the knowledge store is shown. From the parent-child links, we could say that circle is a shape and shape is a Visual Thing. Red and Blue are Colors and Color is also a Visual Thing.



From this representation of partial UKS content, you'll know that at Location1, there is an object which is a Blue Square, at Location2, there is another, at Location3 there is a Red Circle, and at Location4, there is a Blue Circle.

Before continuing with the structure of a Thing, understand that The Universal Knowledge Store, itself, is just a list of Things; Things that contain links to other Things within the Knowledge Store. That means there is no limit to the number of independent UKSs you can have and any subset of a UKS can be treated as a UKS in its own right.

The parent-child relationships of the UKS do not necessarily represent a formal tree structure because Things may have multiple parent Things, there is no exclusion of circular references, and not all the Things in the Knowledge Store must be interconnected—that is, there could be multiple disconnected trees within a single UKS.

These next two properties in each Thing are great programming conveniences but are a departure from biological plausibility.

Labels

Each Thing has an optional label, so when you look at debug information about the content of the knowledge store, it can make sense. Of course, biological neurons don't have labels, which is one

reason it's so difficult to decode what's going on inside your brain. Without Labels, Things are just lists of Links and it's a tough task to trace the Links to figure out what a Thing is. So, with a Thing labeled "Physical Object" with children which are all physical objects, it's easy to say with certainty that a specific Thing with that parent is a physical object too.

A single line of code can find any Thing based on its label.

```
Thing t = UKS.Labelled("Physical Object");
```

So you can get a list of the children of a Thing with:

```
List<Thing> physObjs = UKS.Labelled("Physical Object").children()
```

and immediately get a list of all the physical objects in the UKS. All the child Things typically have some short labels with a sequence number (like Object1, 2, 3) so when you examine such a Thing, you immediately know it's a physical object and roughly when it was added to the UKS without having to trace any Links.

Values

As I mentioned earlier, there are deviations from the idea that nodes contain no information. There is no plausible way to add text or a precise numeric value to a node built from neurons. The mental exercise of considering a graph where none of the nodes has any content will give you a better insight into how your brain must work.

In a computer, though, to know how to spell a word, we simply store a text string. In your brain, since neurons obviously don't support text strings, there must be an ordered list of links to other nodes which represent individual letters. These Letter nodes must have links to other nodes which define the strokes you'd need to write them, the utterance you need to speak when spelling a word out loud, and a definition of patterns of visual input so you can read them. You can see that the complexity needed to store something as simple as a word in a biologically plausible structure can be daunting.

In the UKS, each Thing can have a "Value" which can be *any* data type—a number, a text string, a vector, a color, etc. The Value has

been used to store a data item needed to make the program run as a shortcut to speed development. For example, to store color information, rather than deciding today how the brain might represent color information, the UKS simply stores the raw RGB levels into the Value and the question of the internal representation of color is deferred to a future development iteration. There are several ways that color could be stored in neurons but deciding which way to do it is not as important as being able to use color in other processing.

The key for these two properties, the label, and the value, is that as UKS applications are developed which rely on them, we know we're cutting corners on the biological plausibility front. This may mean that we may need to re-think the algorithm down the road *or* that conversely, we've implemented a mechanism that can give the *Brain Simulator* an efficiency edge over its biological counterpart.

References

In addition to Parents and Children, each Thing has a list of Links named "References" so a Thing can reference any number of property Things. A physical object may reference Things representing color, texture, shape, size, location, etc. but it also may reference Things which are words and phrases that describe the object verbally. Like parent-child, References are mirrored with the ReferencedBy list. The UKS can easily determine not only the color of a Thing, by searching the reference list for a child of Color Thing, but also what other Things have the same color by subsequently following that Color's ReferencedBy list. I should reiterate that this type of back-reference is a great convenience for software development but cannot be implemented with the biological synapses of a single neuron. It requires multiple neurons and multiple synapses for a brain to function equivalently.

In yet another departure from the biological, References lists may be "unordered" or "ordered". As an example, the attributes of a physical object all exist simultaneously so are unordered. On the other hand, the words of a phrase must occur in a particular order or the meaning may be lost. The shortcut of an ordered list also requires a "currentReference" variable which keeps track of where processing is in the list of ordered references. To speak a phrase, currentReference is initially set to the first word reference in the

phrase and is advanced to the next as needed so the word sequence can be spoken at a reasonable rate. To speak each word, each word Thing's currentReference keeps track of the phoneme which has just been spoken.

For bookkeeping purposes, each Thing also has a "use count" which tracks how often a Thing has been accessed. Also, each Link keeps track of "hits" and "misses". These can be used together to determine which Links are important references and which may be irrelevant. That way, the system can figure out that three sides make a triangle independent of what color it might be.

The UKS and Neurons

Everything about the UKS described so far is independent of any *Brain Simulator* neurons. It's just a data structure that is somewhat biologically plausible. To extend the UKS into the Brain Simulator's neuron domain, an extension called UKSN adds two neurons to each Thing, one representing its input and the other its output. So, for example, neurons firing in the SpeechIn module can be connected to UKSN input neurons and corresponding output neurons can be connected to the SpeechOut module. This way, the UKS is accessible with the speech interface using the *Brain Simulator*'s standard synapse connections.

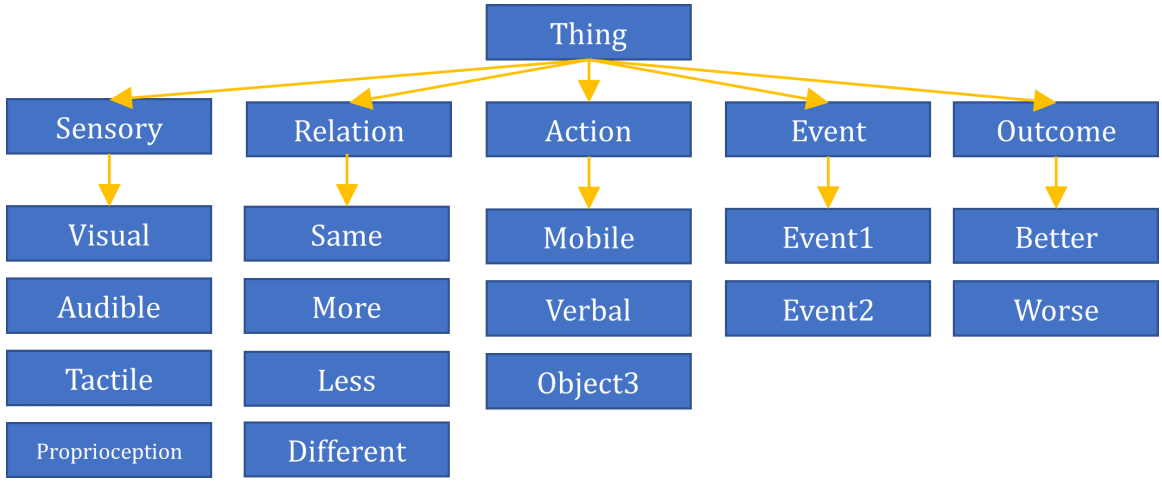
The UKS and AGI

One of the underlying tenets of *Brain Simulator* development is that in order to create General Intelligence, an AGI needs to represent and merge information from widely disparate sources and types. To represent something as simple as "Things fall down," you need to understand about physical things, have seen things fall (an action), know about sequences of actions, have heard and learned the associated words…and on and on.

Although still in its infancy, the implementation of AGI on the UKS can represent all this information in a useful way. As previously described, the organization of data within the UKS is governed by Links, so changing a few Links can completely alter the structure of the information.

With the caveat that the organization of information can be changed easily, here is the current organization of information needed to implement AGI:

- Sensory—information from an AGI's various senses.
- Relation—allows sensory information to be stored. The brain doesn't represent absolute information, it knows that various attributes are the same, different, greater, smaller, and a host of other relationships.
- Action—things that an AGI can do. There will be simple actions (speak a Phoneme) that can be combined into complex sequences (sing a song while dancing).
- Events—combinations of senses such as landmarks (a combination of visual or other inputs) or words heard along with references to actions and outcomes.
- Outcome—the current state of the AGI relative to its internal goals.



The top-level organization of the UKS for an AGI might look something like this. The Sensory and Action areas are fairly obvious as they correspond to known areas of the brain—the sensory and motor cortexes. Events (as described later) are the memories that combine sensory input with an action taken which led to an outcome. They are necessary to determine which action to take next.

The AGI maintains an internal mental model but this isn't so much a Thing as the collection of recent sensory inputs combined with physical positions.

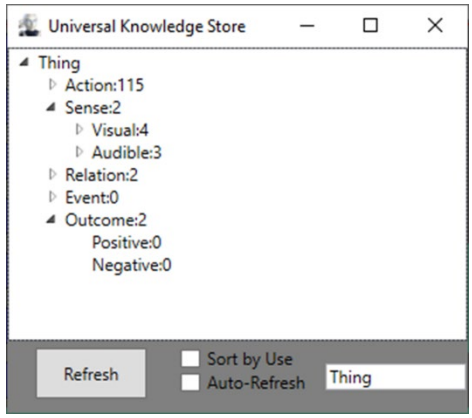
The combination of this information gives rise to the possibility of Reinforcement Learning. In a specific situation, an AGI takes an action that results in an outcome. If the outcome was positive, then

in a similar situation, the AGI will take a similar action. If the outcome was negative, the AGI can choose a different action instead.

The UKS Dialog

To develop UKS applications, you need to know what's going on and there is a dialog that lets you view and drill down into the structure and content of the UKS.

The content dialog can expand lists of children and references. Each Thing's label is shown, followed by the use count and the Thing's value if it has one. When auto-refresh is enabled, you can see the structure, use-count, and values changing as the UKS evolves.



The dialog display of the UKS content shows a tree structure that can be expanded with mouse-clicks to show children and any References. After the label of each Thing is the use-count and the Thing's Value if it has one. In the lower right, you can select any labeled Thing to be the root of the display. Since the structure of information within the UKS is created with Links, it can be modified easily and at any time. With auto-refresh set, the display will immediately update to reflect the current structure and content of the UKS.

Summary and Future Development

The UKS is a powerful general-purpose graph structure that is biologically plausible because it could, potentially, be implemented in neurons. It can store any kind of data and may work in an entirely brain-like way.

The UKS applications demonstrate two learning methods: Correlation and Reinforcement. In the Vision application, Sallie correlates words heard and objects seen to infer which words

describe which objects. In the Maze application, Sallie builds a structure where she can decide which action to take in a given situation to achieve a goal. In both instances, the learning is "one-shot", so a single presentation of information is sufficient. The software could easily be modified so that it would only learn progressively over several presentations—this would make it more life-like but not necessarily better. An inherent advantage of the computer over the brain is its ability to store information, reliably, in a single operation.

The UKS is a prototype that demonstrates the feasibility of implementing such a system. The software within the Vision and Maze applications is specific to the problem and no effort was made to make it general purpose. Now that these applications (and a few others) are working, we can examine the commonality of their store and search functions to find a more generalized solution in these areas as well. Adding these abilities will go a long way toward building a General Intelligence system.

The UKS stores everything. At some point with more complex problems, this will become an issue and a selective forgetting algorithm will be needed.

The UKS is fast enough to support the small demonstrations so far. As progressively more complex problems (and generalized solutions) are addressed, we can assume that performance will become a bottleneck. As the current implementation is single-threaded, it doesn't benefit from multicore or multicomputer operation. There are several ways to address this issue.

The early implementation of the UKS in neurons gives an indication of the maximum scale of the human neocortex. With the neocortex's 16 billion neurons, using the NeuralGraph's 16 neurons per node puts an absolute maximum capacity of one billion nodes. Using a more plausible 100 neurons per node yields 160 million nodes. Observing that the neocortex is not 100% devoted to this form of storage, it is reasonable to assume that the neocortex is limited to fewer than 100 million Things. While this might sound like a lot, from a computer/data perspective it isn't much data at all.

On the other hand, it may be possible to demonstrate AGI with far fewer nodes. If your UKS can only represent 1 million nodes, for example, could it manifest AGI? Obviously, if it's a million nodes of

knowledge specifically about French Literature, then no. But if a one-million node UKS has more general knowledge, then perhaps. Relative to an average person, it would have a limited vocabulary, a limited number of objects it can recognize, a limited number of possible actions and interactions with those objects, a limited ability to plan. Nonetheless,it might still seem intelligent.

That's the intent of the UKS. All-in-all, the UKS forms a powerful platform for AGI research.

Video Links

"Representing Knowledge in Neurons"
http://futureai.guru/videos?id=116

"Introducing the Universal Knowledge Store Pt 1"
http://futureai.guru/videos?id=121

"The Universal Knowledge Store Pt 2"
http://futureai.guru/videos?id=122

Chapter 12: The Simulator, Mental Model, and Planning

The real world is complicated. The way our senses receiving information, there is always ambiguity, errors, and noise which cloud our interpretation of reality. There are no programs today which can interpret input from the real world as well as the human brain—we can expect that to change soon. In order to move forward with software development, rather than coping with somewhat random, non-repeatable inputs from the real world, the *Brain Simulator* includes simulator Modules which can provide input with any desired level of complexity with noise-free, unambiguous, repeatable input.

One impediment to interpreting real-world input is that your brain doesn't just look at its input but builds an internal mental model of your surroundings. Your brain has to work hard to create and maintain that model, but in a computer, it's much simpler. In the same way that the nearly 60 billion neurons which control your body's movement can be replaced with a few microprocessors, building an internal mental model is much easier with a computer than with neurons.

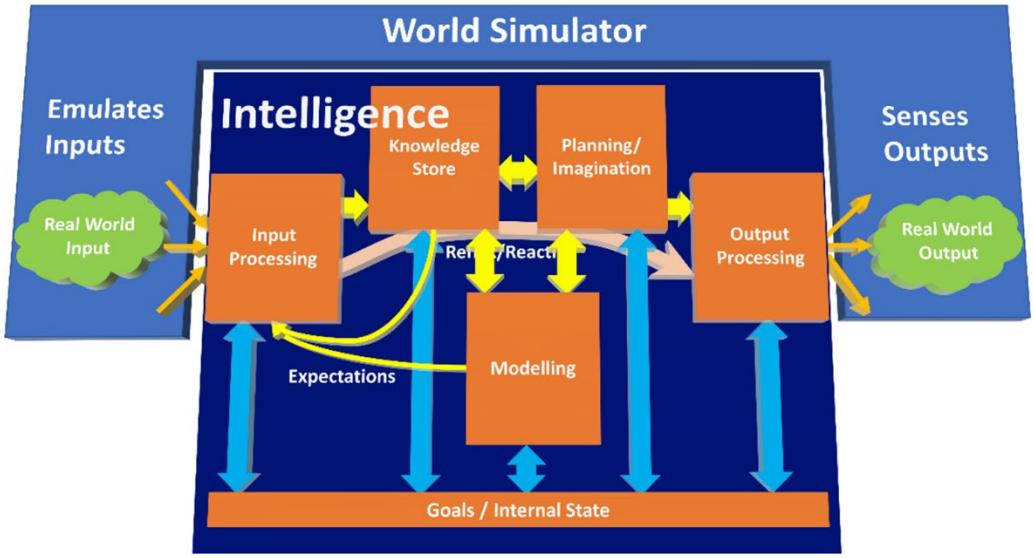
The key is that building on the abilities of the UKS, building an internal mental model is straightforward. Similarly, the planning process can involve modeling so it's built as an extension of the mental model. Other types of planning are also just extensions of the UKS.

This chapter details how these functions are currently implemented in the *Brain Simulator*.

The Simulator

The *Brian Simulator* contains two simulation Modules, a 2D simulator and a 3D simulator. The reasoning is that if Sallie can't

comprehend the simpler 2D world, she won't comprehend the 3D world either so it's best to start with the simpler environment. In both simulators, Sallie can move about and "see" objects in her visual field. The simulator receives information from Sallie's physical functions and updates the visual inputs accordingly.



The Simulator surrounds Sallie. At one end, it can create information for all of Sallie's inputs (senses). At the other, it receives Sallie's actions and updates the simulation to fit. For example, Sallie can move a physical object and this motion is reflected in the touch and vision inputs.

The 3D simulator uses the computer's graphic capabilities to project what Sallie would see from any given position and orientation. The simulator currently supports flat, rectangular objects. No collision or object motion is currently implemented.

The 2D simulator has received much more development effort. It detects collisions between Sallie and the physical objects in the environment and these can not only move the objects but provide input to Sallie's various touch sensors. The 2D simulator includes its own physics to support collisions between objects so if Sallie moves objects there is some representation of friction and a center of mass. Further, if one moving objects collides with another, the second object will move as well.

The only physical object type the 2D simulator supports is a line segment. This simple object is sufficient to exercise numerous intelligence features. Sallie can plan routes, move objects, create structures, and a myriad other possibilities. Each physical object has properties of color and aroma and some objects can move on their own (think birds). When Sallie touches an object, she receives accurate information about it's relative position and orientation and whether she is touching near its end. Aroma, on the other hand, generates a field which declines over the distance from the object. This feature has been used to emulate food. Sallie can follow the direction of increasing aroma strength to get to a destination. To be compatible with the 2D environment, special 1D retina Modules have been created. These can use binocular vision to estimate this distance to visual objects.

For object input, the 2D module has an abstract array of neurons and objects are represented by synapses connecting neurons. To manually add an object to the environment, simply add a synapse.

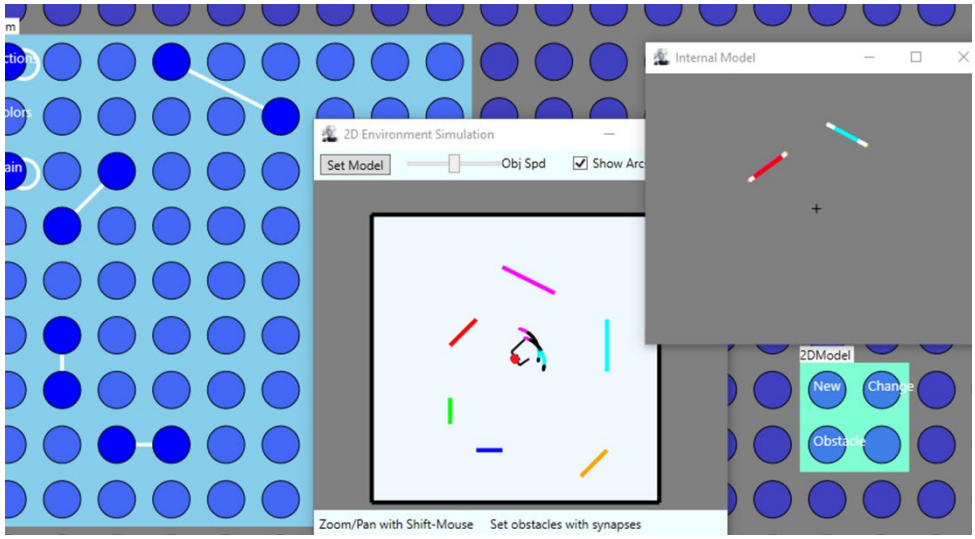
This simulator can also create audible input. Currently, this is in the form of phonemes which describe objects in Sallie's visual field. With this input, Sallie can learn to associate words with colors. With the simulator's feedback, Sallie can learn by imitation to put together intelligible sequences of phonemes to create words and so verbalize about what she sees.

Sallie will appear to "comprehend" the limited 2D environment if she can move objects around to accomplish a goal and verbalize about her actions. To do this, she must have learned about the physics of objects within the simulator, have the ability to plan for a future, and have the ability learn from her own trial-and-error experimentation—much like a toddler. Once Sallie has mastered the 2D environment, spicing the 2D simulator's features into the 3D simulator will be straightforward. Likewise, eventually replacing the 3D simulator with cameras and microphones on a mobile platform is the logical next step.

The Dialog

The 3D simulator dialog shows the visual field as Sallie would see it.

The 2D simulator dialog is much more sophisticated. It shows Sallie's position and orientation within her environment. Sallie's two mobile arms indicate the positions of her touch sensors.



The 2D Simulator dialog box shows Sallie's location and orientation in her environment. The array of neurons to the left shows how the white synapses are used to create physical objects in the simulator.

At the top of the dialog, "Set Model" is a shortcut which places the object information directly into Sallie's internal mental model (see below) skipping the visual system. This is useful for testing to eliminate the errors and ambiguities inherent in the vision system. Two arcs (optional) represent the information in her field of view as the information going to her two retinae.

The "Obj Spd" slider controls the speed of objects which have inherent motion. To make an object mobile, set its synapse weight to a value less than 1 with a positive weight moving in the X direction and a negative value moving in the Y direction. With the speed slider centered, there is no motion. Setting the slider to the right sets mobile objects moving in a positive direction and vice versa.

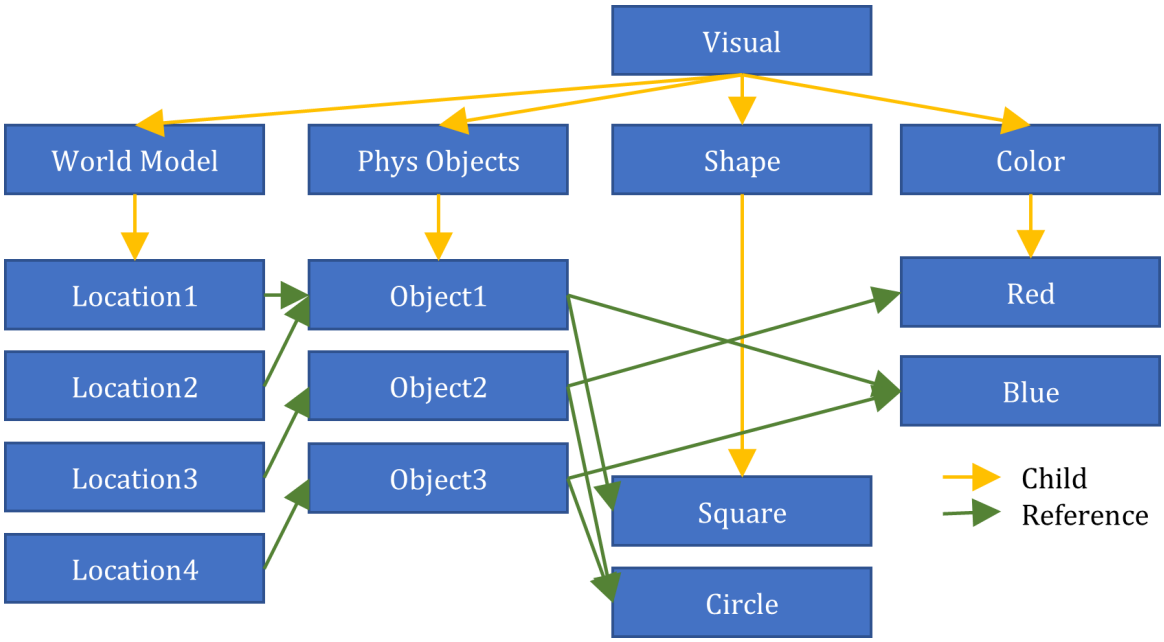
A right mouse-click can direct Sallie to move to a specific position by setting a target position in the ModuleGoToDest Module. This is used in the "Imagination" Network. If the Module does not exist in the current network, right-clicks are ignored.

The Internal Mental Model

What you see isn't the input from your eyes, it's the content of your internal mental model. If your eyes dart around, you don't perceive motion in objects around you. You assume that surroundings are generally static as you move through surroundings. At the same time, you can have a pretty good idea of what's behind you without looking. The level of detail is not nearly as good as it is within your visual field but you certainly presume that objects continue to exist even when you can't see them.

Given that the UKS can store any kind of information how do we store an internal mental model and how does this compare with how your brain does it? If you have a UKS filled with abstract knowledge about objects and their relationships, (perhaps books), how do we instantiate a specific book and place it in some specific location relative to ourselves. You can see a book in front of you. Now turn around. You'll know the same book is behind you, you'll know about what it looks like and how far away it is.

The key is to allocate a new object for each instance of a book which you can see. Each book object can have additional attributes like spine color and the key to modeling is that it carries a location as one of its attributes. These instantiations are children of a Thing labelled "Phys Objects". In the event physical objects have a specific location, the location is a child of World Model. I'll cover how locations are stored and manipulated momentarily. The location is always relative to a center point which is the point of view of the AGI.



This diagram shows how the UKS can represent a world model with instances of physical objects.

Now, for a list of all the objects you currently know the location of, you just need to enumerate the children of World Model.

Moving and Rotating

Thus far, everything has been properties of the UKS itself. But as the AGI moves through its environment or changes it orientation, all the locations need to be updated accordingly so they will remain correct relative to the point of view of the AGI. This is done with just a bit of trigonometry. All the locations which are children of the World Model are updated with every motion.

The PointPlus

As I mentioned in the UKS description, every Thing can have an arbitrary data object attached to it and locations use the PointPlus object. This is a point object which can be accessed or updated equally in either cartesian or polar coordinates. So to rotate the world view, you need only add to or subtract from the direction of every location in the world model. To move (forward) you need only subtract from the X coordinate of every location in the world model.

Internally, the PointPlus keeps both types of coordinates but only updates them as required. That is, if the polar direction is changed,

the cartesian coordinates are not updated until the values are needed and vice versa. This means that repetitive direction changes or motions can be handled with simple addition without any trigonometry. Trigonometry is only needed if rotations and translations are interspersed. Given the relatively small number of objects which must be maintained in the world model, the amount of computation is insignificant.

For current experiments, the relative direction is represented by a single horizontal angle and so is a 2D position. The direction could easily be extended to be a horizontal direction and an elevation angle so as to represent 3D positions.

The angle to any object is based on a specific pixel location in the visual sensor and so have consistent accuracy. But because distances are estimates based on binocular vision, their accuracy decreases with objects which are further away. The PointPlus also carries a "confidence" value which represents the expected error variability of the distance value. As the AGI sees an object again, it is likely that the distance value will be slightly different. The confidence value is used to determine whether the internal value should be updated to the new, sensed value. That way, if an object is seen at a large distance, there is likely to be a significant error and the confidence is low. As the AGI moves closer to the object, new values come in with better confidence and the internal values are updated. As the AGI moves further away, new incoming values will have a lower confidence so the internal values are *not* updated. Values received from the touch sensors have the highest confidence.

Imagination

There are two ways that the Internal mental model can be used for imagination.

1. Imagining a different point of view
2. Seeing "imaginary" objects

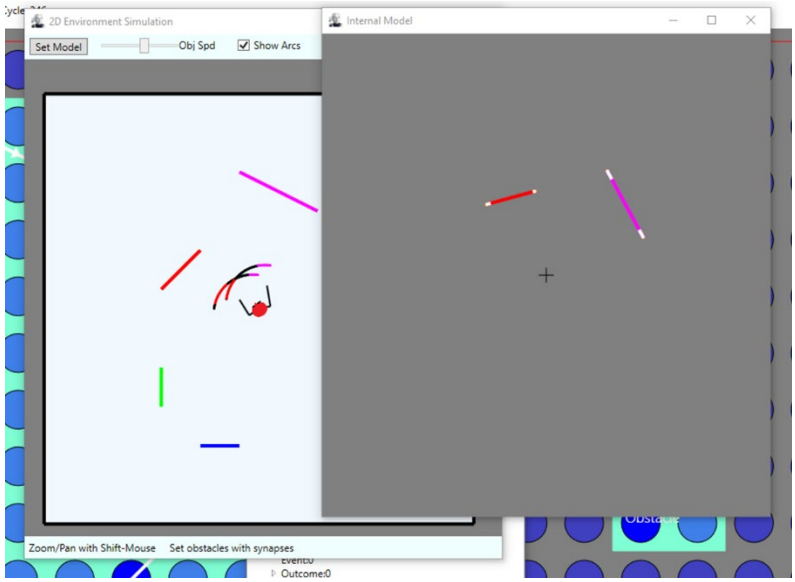
Both these mechanisms have been implemented in the internal model. The first is used in the Imagination network as Sallie imagines what she would see from various alternative points of view and moves to that point if she can see her destination from there.

The second has been implemented and will be used for planning as that process is generalized.

The Dialog

The dialog associated with the Module2DModel displays the current content of Sallie's internal mental model from her point of view. As Sallie moves and turns, the content of the model is updated and displayed so that objects in front of Sallie are always upwards in the dialog display.

Each segment is shown in the color Sallie has seen and is displayed with white ends which indicate the confidence or accuracy in the position of the object. Because Sallie's vision emulates binocular vision, the angular accuracy of any point in the model is constant but the distance accuracy degrades substantially with distance to the object. Longer white portions indicate lower accuracy. If an object has been touched, this is the most accurate possible position and there will be no white portion.



The dialog for the Internal Model Module shows the physical objects Sallie knows about and are shown from her point of view. In this case, there are objects in the simulator which are not in the model because Sallie hasn't seen them yet. The white ends indicates the confidence/accuracy of the distance value to the objects. The longer white ends on the right-hand object indicate lower confidence because the object is further away.

Planning

Planning involves internally simulating a sequence of different possible actions in order to achieve a goal. But first, you need to learn about the individual actions and their consequences. The development of this process in the *Brain Simulator* is still in its infancy and is used in the Maze application (below).

The key is learned abstract structures within the UKS which tie together a situation, an action taken, and the outcome. If these are taken individually, an organism can evaluate its input, find the best match among its stored situations, then choose the action which led to the best result. In this manner, an entity can learn any number of situations of any desired complexity and relate them to actions, also of any desired complexity. This simple process can explain the behavior of animals which can be trained to respond to relatively complex commands with relatively complex behaviors.

The process can be extended by adding the mental mechanism so that a *sequence* of actions can lead to a goal. So if a chimpanzee knows that standing on a block raises it up, and knows that stacking one block atop another creates a taller block, then it can envision stacking multiple blocks in order reach the bananas at the top of the enclosure.

The implementation of planning within the *Brain Simulator* does will be expanded to offer this level of generality. Current solutions are purpose-built for a specific application but the UKS structure and internal model are intended to allow for this generalization.

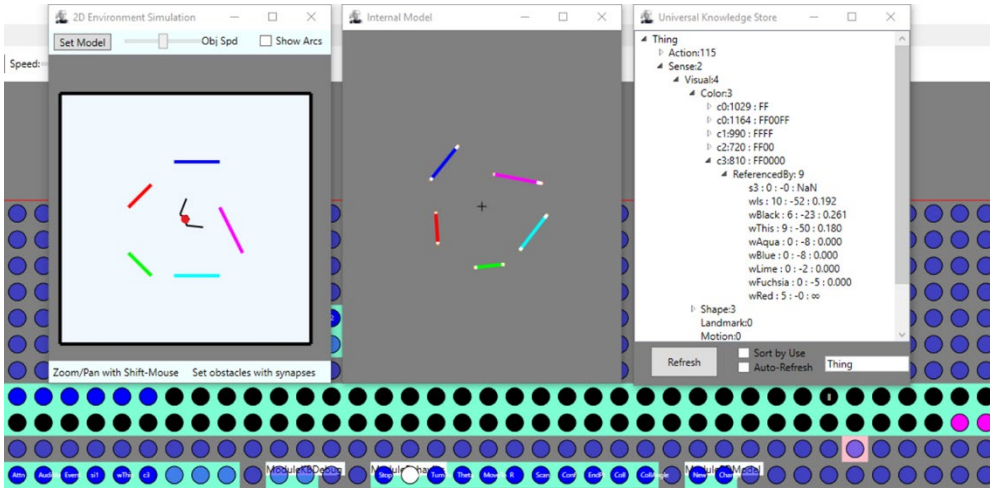
Application 1: Vision, Associating Words and Objects

Two prototype applications have been created to illustrate how the UKS can be used to build intelligent behaviors. In the first, Sallie can be directed to move freely around her simulated environment and look at Things. At the same time, simulated voice input periodically says the color of the object she is looking at (e.g., “This is red.”). While other Modules process the voice and vision functions, the focus here is on how the UKS is used.

As Sallie moves about the environment, objects she sees are interpreted into segments with a color that she can see. These are stored in the UKS in terms of Segment Things (children of “Segment”

which is a child of "Shape"). Each segment has references to two Point Things (also children of "Shape") and a Color Thing (a child of "Visual" which is a child of "Sense" along with "Audible"). Every point she sees can be represented in terms of an angle (from straight-ahead) and a distance, which is estimated from her binocular vision. These are stored as PointPlus values on the Point Things.

As she moves and rotates through her environment, all the PointPlus coordinates can be updated with her motion so they are always up-to-date. Sallie can easily determine if an object she sees is the same as one she has seen before.



In this view of the Brain Simulator*, the left window shows the Environment Simulator and Sallie's position within her environment. The center window shows Sallie's internal mental model of the world from her point of view; the display represents the UKS content of shapes and colors. The two rows of neurons below represent Sallie's visual field; she can see blue at the left and magenta at the right. The right-hand window shows the content of the UKS including colors which she has seen. Color c3 is expanded so you can see that it is referenced by the shape s3 and has correlations with words she has heard. You can see that the word wRed has the best correlation with the color c3.*

This forms the basis of Sallie's internal mental model. It's not so much a Thing, but the collection of recently-seen objects. All the points she knows about automatically update themselves as Sallie moves and rotates within the environment. There is an optional display window of Sallie's internal mental model, but it simply

accesses the current state of the Things in the visual memory portion of the UKS. It displays them so the AGI developer can gain insight into Sallie's internal mental state. You can see the same data changing within the tree-view dialog box but it doesn't make as much sense as the graphical display.

The Environment Simulator periodically speaks to Sallie, telling her the color of the object in front of her with the phrase, "This is [red]". This is added to the UKS as Phrase Things, which have ordered references to Word Things. Because Sallie can see more than one object at a time, there is some ambiguity in the announcement, and the words "This is" are extraneous to the meaning. Over a period of samples, Sallie can correctly associate color names she hears with the colors she sees. She learns that the "This is" is irrelevant because it is heard with everything and so has no differentiable meaning. Whereas the color names associate specifically with different objects she sees.

Application 2: Maze / Learning by Trial and Error

There are plenty of ways for a computer program to solve a maze. This approach is interesting in that it utilizes the UKS and builds a structure within the UKS that can be generalized to a wide variety of intelligent behaviors.

This maze in Sallie's environment illustrates how the UKS can be used to keep track of landmarks. The Event/Action/Outcome triples stored in the UKS form the basis of reinforcement learning.

The content of the maze is programmed directly to the UKS in the structure of segments, points, and colors as described above. This is

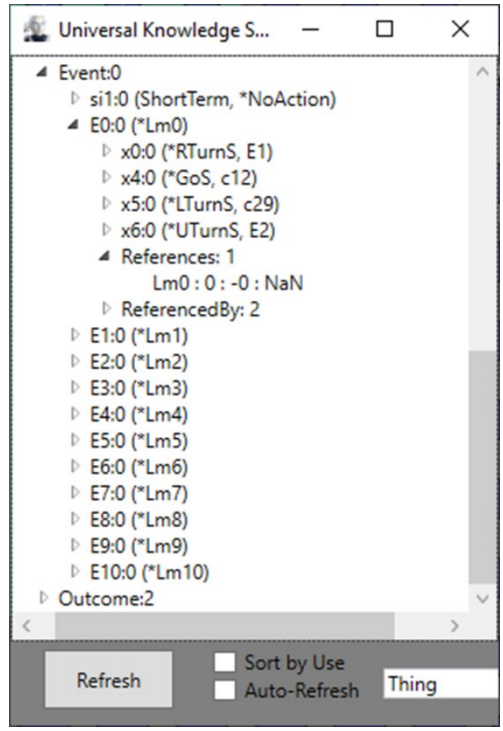
done so that possible vision errors can be excluded from the behavior process. Further, the maze is orthogonal so at each decision point, a limited set of action choices is possible.

As she explores the maze, Sallie remembers "Landmarks" which are Things that reference nearby segments. Unlike the segments in the previous demonstration, these Landmark Segment positions are static and do not update their position when Sallie moves about. That way, Sallie can know when she has returned to a landmark where she has been previously because she can recognize the fixed objects.

At each landmark where a decision can be made, Sallie creates an Event Thing that references the Landmark. Children of the Event Thing are action/outcome pairs—each references an action Sallie took and the outcome which subsequently occurred. She then takes a random Action (RTurn, LTurn, GoS, UTurn) and creates the action/outcome Thing as a Child of the Event Thing. She then proceeds until she reaches another possible decision point or is blocked at a destination.

If she is blocked, she remembers this color as the outcome of the current action/outcome pair. She makes a UTurn and continues on. If she instead reaches another new decision point, she creates another Event Thing in the UKS and uses that as the reference. If she reaches an Eventshe has previously visited, she can select a random action from those she hasn't tried before and continue the process.

Eventually, she learns all possible actions for all events. At that point, the entire maze has been traversed and the event list is complete.



This view of the UKS after Sallie has explored the maze shows that Sallie has encountered 10 decision points or "Events." E0 is expanded so you can see that it is related to landmark Lm0 and at that point, Sallie tried four different actions. For example, she took a right turn which led to the event E1 and she went straight and reached the goal color c12.

Now, given a goal color, the UKS can be searched for all Event Things that have a child which references the goal color. These events are also searched (similar to using the Recur relationship) until eventually the current Event (where Sallie is now) is encountered and Sallie takes the action associated with the Action/Outcome pair which led toward the goal.

Here are a few important component points to consider:

1. Everything needed to create data structures of essentially unlimited complexity exists in the basic UKS structure. Recall that the UKS can be implemented in biologically plausible neurons.
2. What I've called the action/outcome pair is actually a triple when you take the Event into account; an event/action/outcome triple. In general, any intelligent entity considers its current situation in terms of similar situations from past experience and

then takes the action which leads to the outcome that best approaches its current goal.
3. Finding your way based on recognizing landmarks is a biologically likely approach.
4. The brain can use event/action/outcome triples repetitively to achieve a goal but usually can only consider a few steps toward a goal. The computer has the tenacity to consider unlimited steps to achieve its goal.
5. The Event Thing can be as complex as necessary. In this implementation, it is limited to being a landmark consisting of a half-dozen segments because of limitations in the maze design. In the brain, not only could landmarks be significantly more complex but the event could consist of any number of diverse factors…anything the mind can sense or feel could be inputs to an event.
6. Similarly, since the action and outcome are Links to other Things, these could also be significantly more complex.
7. The maze in this demonstration looks pretty simple because you can see it all at once (as from above). If you instead put yourself in Sallie's position and think of this as, perhaps, a hedge maze, it would tax the limits of the human brain's abilities. If there were a unique goal for each endpoint, I, certainly, would be unable to go directly to any goal endpoint from any other without error. This maze contains only ten decision points (Events).

The structure of Event triples can be generalized to form the underpinning of what is called "Reinforcement Learning." In all situations, you remember what action you took in that situation and what outcome you received. The outcome might be simplified to a reward or punishment but in general, the situation, the action, and the outcome can all be complex. Subsequently, this structure would be pruned to eliminate situations that are similar, actions that led to indifferent results, etc. Alternatively, results that represent significant improvements or detriments could be made more important.

Video Links

"How Sallie Learns & the Universal Knowledge Store"
http://futureai.guru/videos?id=123

"How Sallie learns with Reinforcement Learning"

http://futureai.guru/videos?id=124

"Short: Navigating a Maze"
http://futureai.guru/videos?id=140

Chapter 13:
Brain Simulator Performance on Multicore and Multiserver Systems

This chapter is adapted from an academic paper and contributes to an estimate of the computer power required to emulate the entire human neocortex with the *Brain Simulator II* by implementing and measuring the performance in a single multicore computer and in a cluster of networked computers. The results are extrapolated to the scale of the neocortex based on measurement of the computational performance on the single machine and the network traffic needed for server-to-server transfers. Algorithmic improvements are identified for future implementation.

The spiking neural model is based on observations of biological neurons and differs from most ANN algorithms in two important ways: 1) the array of synapses for any neuron is sparse and 2) significant processing is only needed for neurons that fire. These both contribute to the performance achieved on a single CPU which is RAM-speed limited. On the other hand, the sparse synapse array makes this algorithm less amenable to GPU acceleration.

In general, computational performance scales linearly with the number of active synapses because the number of synapses is large relative to the number of neurons. Importantly, although computational and RAM requirements scale linearly with the number of synapses per neuron, network data requirements for machine-to-machine transfers generally scale with the number of neurons simulated on an individual server.

As the performance is directly related to the number of synapses per neuron, we'll examine the relationship between the number observed in the biological brain vs. the likely number needed in the simulation. The computer can allocate new synapses quickly while the brain cannot (both can adjust weights quickly). This means the brain must include a large proportion of near-zero-weight "synapses-in-waiting" to be used when the need arises. The computer need not simulate these because they can be allocated quickly when needed.

The overall conclusion is that a model of the complete neocortex could be implemented on today's hardware. The specific number of machines required depends on the number of synapses per neuron, the complexity of the neuron model, and whether the intent is to emulate in real time, or slower or faster by some factor. A sample calculation is done for 160 servers.

Background

While focusing on the performance of algorithms in multicore and multicomputer configurations, some neuroscience information is necessary to describe the scope of the problem. Overall, the brain will exceed the performance of any single CPU for the foreseeable future, so this chapter estimates the issues in processing across multiple parallel CPUs.

Throughout the chapter, it should be noted that most biological measurements are approximations with only one or, at best, two significant digits. This section also describes values selected for subsequent estimates to help define the scope of variability in the estimates.

Neuron Function

As described previously, the neuron is essentially a digital device in that neural spikes are about the same size and variations in spike shape are considered noise. Relative spike timing is usually considered its only variable feature, and this is also subject to a great deal of noise (jitter). The amount of charge contributed by a synapse is limited to approximately 100 discrete values [Montgomery]. Although neurons are often described in terms of the continuous mathematical functions relating to membrane diffusion, exponential charge decay, etc., discrete approximations for these functions are

used in this presentation which likely exceed the accuracy of biological neurons because of the high noise levels in the brain [Faisal].

Neuron Performance

Although the function of a neuron can be measured electronically, it is misleading to think of the neuron as an electronic device. Instead, it relies on the physical transport of ions or changes in their orientation and thus works in timeframes of milliseconds—a billion times slower than today's electronic components. The maximum expected firing rate for a neuron is about 250 Hz but this is not sustained, as neurons in the neocortex are estimated to fire only once every six seconds on average [Grace, Lennie]. This low average firing rate will be important in calculating the number of neurons that fire vs. the number emulated on a single server.

The length of the axon is variable and, in neurons that transfer signals to the human body, may be over a meter in length. Within the brain, axon lengths can be loosely grouped into "long", with lengths averaging 100 mm, and "short" with lengths averaging 10 mm [Braitenberg]. This categorization will be important in estimating the number of axons in a computer model which cross a boundary between one physical computer and another.

Nerve conduction velocity for unmyelinated short axons is also quite slow at just a few m/s (walking speed). This means that the signal propagation from the cell body to the destination synapses may take several milliseconds, and this should be taken into account when estimating the necessary timescale resolution of the simulation. In estimates, a 2 ms per neuron cycle is used.

Learning in biological neurons is not fully understood, although Hebbian learning is known to adjust synapse weights based on the near-concurrent firing of connected neurons. Other learning mechanisms may also exist but learning only affects a tiny portion of synapses at any given time. For example, once learned, the synapses involved in reading or understanding language cannot change substantially or one could forget these abilities rapidly if they were not used/reinforced. While one might learn new words, most learned words, the recognition of characters, etc. are seldom modified.

Useful Synapses

At its destination, the biological axon branches out into as many as 10,000 synapses. The number of synapses that must be emulated will be smaller than the number measured in the neocortex by a substantial factor for several reasons.

The computer can allocate new synapses quickly while the brain cannot. Biological synapse weights can be modified in tens of milliseconds while synapse creation and migration happen over periods of hours and days. This means the brain must include a large number of near-zero-weight "synapses-in-waiting" to be used when the need arises by adjusting the weight. The computer doesn't need to simulate these because additional synapses can be allocated quickly when needed.

Future confirmation of this hypothesis and the value of this factor could be estimated from the distribution of synapse weights within the neocortex, which is not presently known. It is also likely (but not yet observed) that multiple parallel synapses are needed to create an effective high synapse weight (again, the distribution of synapse weights would be useful). In a simulation, these multiple synapses can be consolidated into a single synapse with a weight equal to their sum.

For the simulations, a factor of 100 is used, meaning that instead of 10,000 synapses per neuron, only 100 are simulated. The effect of this factor is clearly stated so adjustments can be made easily to the overall estimates.

Further, for any of these synapses-in-waiting, separate synapses are required for those which are potentially excitatory and those which are inhibitory. These different synapses act with different neurotransmitters and cannot easily shift from one to the other. Since the computer can easily change the sign of a synapse weight from positive to negative, these multiple synapses are not needed.

Although not addressed in this chapter, a similar factor may likewise be applied to the number of neurons to be simulated. As mentioned in the Universal Knowledge Store chapter, we speculate that 100 neurons are used in the brain where we can only identify 16 as being needed. The brain may contain many redundant neurons for reliability while the computer may be able to ignore these and simulate just one because it is much more reliable. Further, the

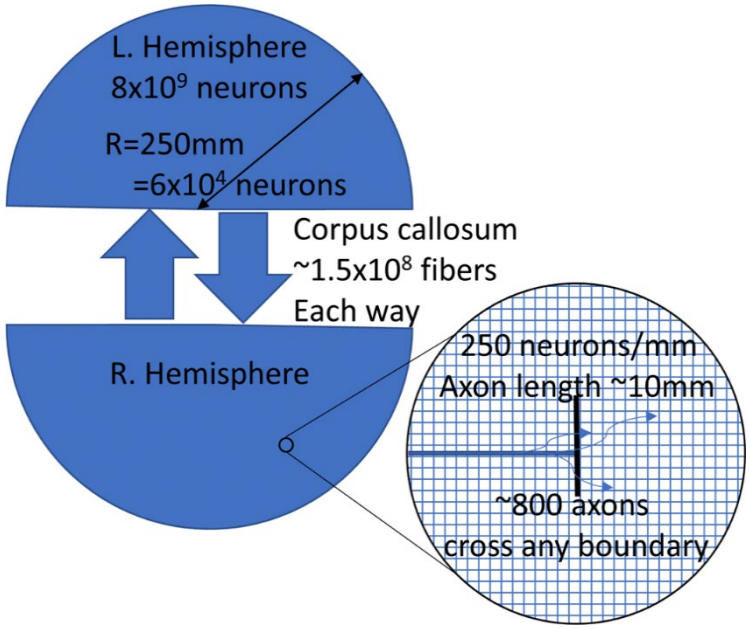
computer may be able to simulate clusters of neurons easily to eliminate the need for substantial numbers of individual neurons. One might conclude that a full neocortex simulation might be accomplished with many times fewer neurons than the brain possesses.

The Brain's Connection Count

The human brain can be considered in three parts: the brainstem which is largely responsible for autonomic functions; the cerebellum which is responsible for muscular coordination; and the neocortex which is responsible for higher-level functions. This chapter will focus specifically on the neocortex.

The neocortex contains about 16 billion neurons which are concentrated near the convoluted outer surface while the interior consists of a mass of axonal connections. If smoothed out, the neocortex would roughly be a disk with an area of 2,600c m^2 (a 250 mm radius) as shown in the figure. In the neocortex, the neurons are in several layers near the surface but for the purpose of these calculations, the layering can be ignored with all the neurons assumed to exist in a single layer.

The neuron density is therefore 16 billion/2,600 cm^2 or ~60,000/mm^2 or (linearly) ~250 neurons/mm. With the average short axon length of 10 mm, we can expect that neurons routinely connect to others that are 2,500 neurons away or more. The flattening of the simulated neocortex will make more columnar axons shorter as they don't leave the plane of the outer layers and then return. But it will model others as longer if they connect from one fold to another. A length of 10 mm will continue to be used for estimation.



The neocortex can be modeled as a disk of neurons with the two hemispheres being largely independent. Each with 8 billion neurons, they are connected by the 300 million fibers of the corpus callosum which represent the ~100mm-long axons of their respective neurons. Within each hemisphere, the number of axons crossing any particular boundary can be estimated by considering a line of neurons forming a perpendicular to the boundary and multiplying by the length of the boundary or about 200,000 axons/mm of boundary length (in each direction).

We can use these factors to estimate the number of axons that cross any given boundary within the neocortex, which will be necessary to estimate the number of signals which will transfer from one computer to another in a multiserver configuration. The likelihood that any randomly oriented 10 mm axon crosses a boundary is given by:

$$\mathrm{Cos}^{-1}(d/10)/\pi \qquad (1)$$

where *d* is the distance from the neuron to the boundary (in mm). This is the portion of a circle of radius 10 mm centered on the neuron which crossed the boundary.

Since the neuron can be anywhere from 0 to the axon length away from the boundary, summing these probabilities along a line of neurons perpendicular to the boundary (as in the inset in the figure) leads to the expectation that any row of neurons will likely present

approximately 800 axons crossing the boundary or 200,000 axons/mm of boundary length.

$$800 \cong \sum_{d=0}^{d=2500} \cos^{-1}(d/2500)/\pi \qquad (2)$$

In a neocortex hemisphere, any radial slice through the neocortex can be expected to be crossed by 50 million axons. This figure will be used to estimate the amount of data to be transferred from machine to machine if the neocortex were subdivided into multiple sectors. Long connections serve to increase the data transfer requirement.

The Simplest Neural Algorithm

The simplest neural algorithm is "Integrate and Fire" [Abbott] which is given by equations 3, 4, and 5. Numerous features could be added which make the algorithm more biologically accurate [Gerstner] as will be discussed later.

The algorithm is split into two phases so the result becomes independent of the order of the neuron calculation and is more amenable to parallel computation. In the equations, time $t+$ (calculated in Eq. 3) is the intermediate time between t and t+1. u_{t+} represents the intermediate value which is calculated for each neuron. In the second phase (Eqs. 4 & 5), the internal value is updated for all neurons and if the threshold ϑ has been reached, u is reset to zero and a spike is transferred to the output.

$$u_{t+} = MAX\left(u_t + \left(\sum_i w_i O_i\right), 0\right) \qquad (3)$$

$$u_{t+1} = \begin{cases} u_{t+}, & u_{t+} < \vartheta \\ 0, & u_{t+} \geq \vartheta \end{cases} \qquad (4)$$

$$O_{t+1} = \begin{cases} 0, & u_{t+} < \vartheta \\ 1, & u_{t+} \geq \vartheta \end{cases} \qquad (5)$$

As an example of the issue that this two-phase calculation corrects: if a neuron receives two inputs with weights +1 and -1, the order in which these are processed could change the outcome. In a multiprocessing implementation, the output would be indeterminate. With the two-phase approach, all summing is performed prior to threshold detection so the calculation is consistent.

In the *Brain Simulator*, the algorithm is "inside-out" in that each neuron maintains a list of synapses which are its outputs. When the neuron fires, it adds the synapse weight to the internal charge of each target neuron. A synchronization lock on each target neuron charge value allows for multiprocessor operation without potential race conditions where multiple threads running on different cores might attempt to modify the internal charge of single a neuron simultaneously. In practice, such collisions are extremely rare so these locks are insignificant to performance.

Different from ANNs

It is important to note distinctions between this spiking algorithm and more typical ANN algorithms. This algorithm's neurons output digital spikes as opposed to floating-point numbers. Accordingly, processing is only required for neurons that are firing. Thus, processing time goes up with the number of neurons that are firing and the overall array size contributes only a slight overhead. Based on the fact that a neuron fires only once every 6 s on average and using a 5 ms cycle time, an individual neuron would be expected to fire once every 1,200 cycles. For an array of 100 million neurons, processing is expected for only 83,000 neurons in each cycle. If a 1 ms cycle time is selected, the expected number of firings drops to only 16,000. This selection of the cycle time does not impact the number of neurons firing per second and so the CPU requirement does not scale directly with the cycle time, but the "overhead" processing is required for every cycle regardless of the cycle time.

Further, in this algorithm, synapses of a neuron can connect directly to any other neuron in the network. In the brain, the synapses connect to other neurons within the radius of the axon length (10 mm) so there could be 10,000 connections from a possible 6.5 million target neurons. This still represents such a sparse array that this algorithm is much less amenable to the GPU acceleration favored by ANN algorithms which rely on filled arrays.

The focus of most ANN systems relates to backpropagation learning. For this discussion, learning affects such a tiny portion of synapses in any cycle that it is not included in this performance analysis and so the analysis is feedforward only. Although Hebbian learning has been implemented, it is not included in this analysis.

A departure from biological equivalence in this simulation is that all synapses run direct from one neuron to any other. Because biological synapses are clustered at the end of the axon, improved efficiency may be possible, particularly in a multicomputer implementation.

Performance in a Multicore Environment

In this section, data is presented for processing performance on a single server which can be used in estimating the number of servers for the neocortex simulation and some configuration requirements (RAM, cores, etc.) for each server. All timing measurements are made using the system high-precision clock which presents time in 100 ns increments. Timings were then calculated with a moving average of 100 readings to create repeatable results.

Tests were performed on a 64-core AMD Ryzen 3990X CPU running at 4.0 GHz with 128 GB of quad-channel DDR4 3200 RAM with Windows 10 Pro. Testing was performed on the *Brain Simulator II* version 0.4.

Sensitivity to number of neurons (overhead)

There are two components to the algorithm's processing time that predominate with different configurations of network 1) "overhead" and 2) "neuron processing." As mentioned before, neuron processing is only required for neurons that fire but there is some degree of overhead that scales more-or-less linearly with the number of neurons. It is necessary to keep track of which neurons require processing and this is done with a bit-field with each bit representing a neuron, combined in 64-bit words. This means that with a single 64-bit memory access, the software can determine which of the 64 neurons require processing *if any*. This proved much faster than maintaining a list of neurons requiring processing.

Overhead was measured by allocating neural arrays with no synapses and no neurons firing and is shown in Table I. This area of code has been optimized to minimize RAM access and so is substantially faster with increasing numbers of threads. At this stage of development, it appears that overhead processing is intractable so any real-time simulation requirement is limited by overhead issues. On the other hand, simulating a 2 ms cycle in 20 ms makes overhead insignificant.

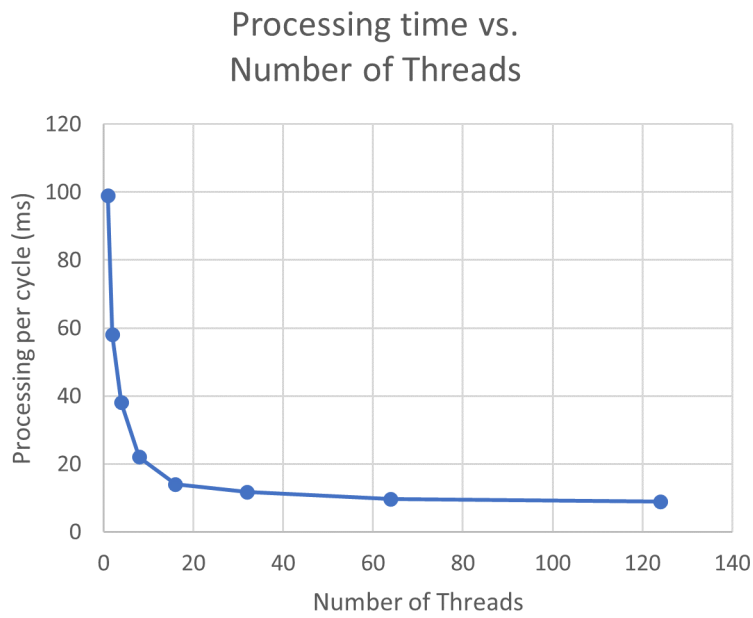
In further tests, overhead processing has not been subtracted out but explains the mixed-slope processing times. Note also that for 100 million and 1 billion neurons, RAM limits on the test server precluded allocation of substantive numbers of synapses per neuron.

TABLE I. OVERHEAD TIMING MEASUREMENTS

Number of neurons	1M	10M	100M	1G
Time per cycle (ms) 124 threads	0.70	1.8	3.7	26
Time per cycle (ms) 32 threads	0.52	1.3	7.6	62
Time per cycle (ms) 16 threads	0.4	0.96	8.4	82

Sensitivity to number of threads

For these tests, an array of one million neurons was allocated, each with 100 random synapses. These arbitrary numbers were chosen to facilitate ease of testing. Random synapse weights were adjusted so that approximately 33,000 neurons per cycle would be firing, which is representative of the number of expected neurons firing in an array of 100 million neurons with a 2 ms cycle time. If one were to decrease the cycle time, fewer neurons would fire in each cycle but overhead processing would become more significant.



This graph shows the observed processing time per neural cycle to handle a million neurons firing, each with 100 synapses set to fire 33,000 neurons per cycle. The total of 3.3 million synapses being handled in 10 ms leads to the raw figure of 330 million synapses/s.

In any neural network, the number of synapses is large relative to the number of neurons and overshadows other factors so that processing time goes up linearly with the number of active synapses.

The 64-core machine is not processor-limited as near-maximum performance is achieved well short of all cores processing fully. Examination of the disassembly with a performance profiler showed that with large numbers of threads, over 90% of the computer time is spent waiting on the single instruction where the CPU must retrieve the target neuron from RAM to add to its charge. Since the target is at a random address relative to the current neuron, nearly every access to a target neuron will result in a CPU cache miss and all CPU cores must wait in line to retrieve their target neuron values from RAM.

A side effect of being RAM-limited on synapse targets is that neuron processing time is essentially irrelevant as long as it depends on neuron values that are in the CPU cache. With a more sophisticated neuron model, such as in [Izhikovich], the CPU will spend time calculating the neuron value which would otherwise be

spent waiting for other threads. As an example, a leakage factor was added which causes neuron charge to decay exponentially. Not only did this not increase processing time, but processing time decreased measurably.

Sensitivity to Synapse distance

It was observed that processing time decreases as "axon" length decreases (neurons are nearer each other in the array) since nearby target neurons are more likely to reside in the CPU cache. As the synapse list approaches a continuous array, a six-fold increase in performance was obtained. This has not been pursued as it is not biologically plausible.

Conclusions for Server Configuration

As currently implemented, each neuron requires 144 bytes and each synapse requires 16 bytes of memory. While the processor requirement goes up only with the number of neurons and synapses that fire, the numbers of neurons and synapses allocated dictate the RAM requirements.

TABLE II. RAM REQUIREMENTS

Synapses/ neuron	1 millino neurons	10 millino neurons	100 million neurons	1 billion neurons
10	304 MB	3 GB	30 GB	304 GB
100	1.7 GB	17 GB	170 GB	1.7 GB
1,000	16 GB	160 GB	1.6 TB	16 TB
10,000	160 GB	1.6 TB	16 TB	161 TB

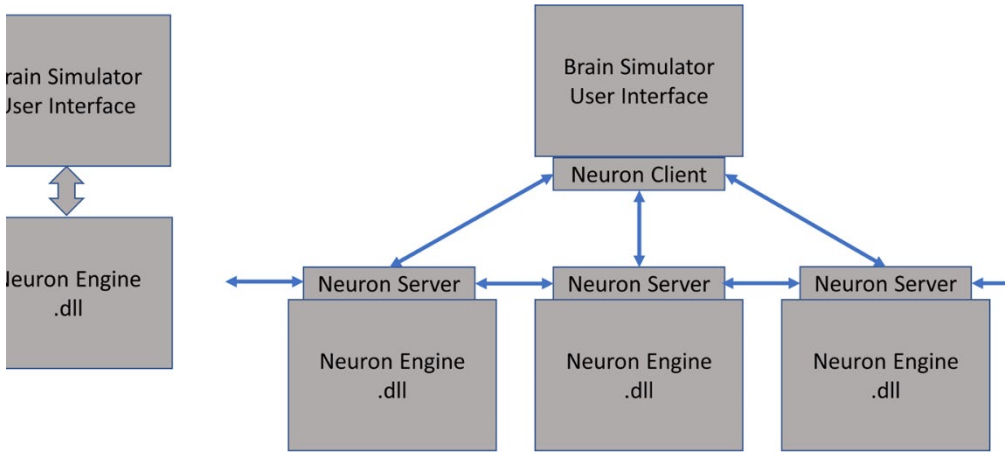
As the system performance is RAM-access limited, the shaded areas of Table II would be useful. Further, the performance improvement for more than 16 cores (32 threads) is marginal.

As previously estimated, a server with 100 million neurons and 100 synapses per neuron would be expected to process 33,000 active synapses per 2 ms (real-time) cycle and would execute cycles in about 12 ms (10 ms measured +2 ms estimated additional overhead). Accordingly, the server would be running at one-sixth real time. Any number of tradeoffs can be made but in general, the processing time will decrease with decreasing active synapses.

Performance in a Multi-Computer Environment

This section presents results of initial experimentation to establish the performance characteristics of a multicomputer implementation, while the following section projects these results to a complete neocortex emulation. Here, we consider the ability to handle larger arrays of neurons without a prohibitive loss in performance.

For multicomputer testing, two additional computers were used: An Intel i7 4565 CPU running at 2.4 GHz 16GB DDR4 dual-channel RAM running at 1,198 MHz, and an Intel i7 6700 running at 3.68 GHz with 16GB of dual-channel DDR4 RAM running at 1,064 MHz. Note that these computers are substantially slower than the one used in the previous section. All computers are connected with a 1 Gbps ethernet LAN.



In a single-computer configuration (left), the user interface communicates with the server engine directly through RAM. In a multicomputer configuration (right), the same user interface and engine communicate through a LAN with thin client and server wrappers. Neuron servers send synapse firing information directly to each other. Although Neuron Servers can communicate directly with any other server, in this experiment, all synapse connections are "short" and will target an adjacent server.

Each server runs the same Neuron Engine .dll as in the previous tests as shown and the Neuron Server layer handles synapse references that extend outside the array on the local machine ("boundary synapses"). When a boundary synapse activates, its weight and destination are placed in a queue. When the basic

Neuron Engine cycle is complete for all local neurons, boundary synapses are dequeued and sorted so that firings can be clustered into data packets and sent to the correct server.

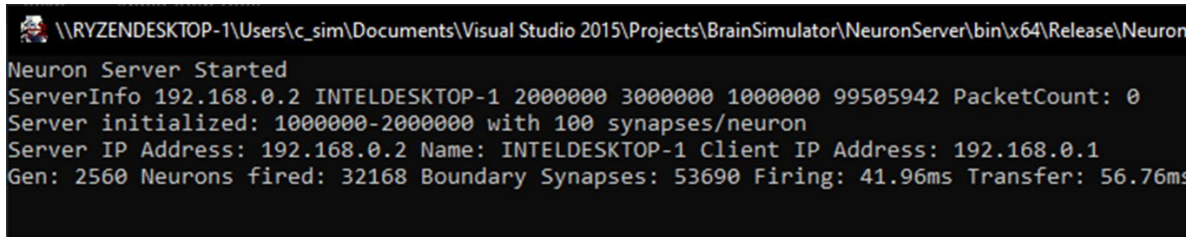
On the receiving end, each server listens for incoming packets and makes the appropriate changes to the target neuron internal charges. No significant effort has been expended in optimizing this process as it is assumed that the data transmission time will overshadow any computation time. For example, the encoding/decoding process is single-threaded. This "data transfer" phase was added for ease of development and measurement and significant possible performance improvements are outlined later.

In this initial implementation, the client directs all servers to execute a single neuron cycle and then waits for all servers to complete the neuron cycle and then transfer any boundary synapses with timing results shown in Table III. Because of the synchronized nature of this implementation, the system runs at the speed of the slowest computer in the network. This issue could be avoided by using a cluster of matched, high-performance servers.

TABLE III. TIMING FOR MULTIPLE SERVERS.

Number of servers	Total Neurons	Total Active Neurons	Overall cycle time	Timing	Total boundary synapses
1	1 M	0	10	1.5/0	0
1	1 M	34,000	88	82/0	0
2	2 M	63,000	116	55/53 48/44	99K
3	3 M	93,000	115	51/49 44/50 11/47	146K

Table III shows that after the first server, cycle time is independent of the number of synapses because the number of boundary synapses is constant for each added server. The "Timing" column shows the firing and transfer times for each server. These can be subtracted from the overall cycle time to estimate the overhead of running in a client/server configuration.



Each Neuron Server reports performance data including the amount of time spent in the firing algorithm vs. the amount of time in data transfer along with the number of active boundary synapses.

Each server can transmit approximately 50,000 boundary synapses in 50 ms or ~1 million synapses/s. Each boundary synapse requires 9 bytes of information, the target neuron, the weight, and a flag. These are packed into UDP datagrams with a maximum of 1,500 bytes (the default maximum packet size) so each datagram packet can send 166 active boundary synapses. UDP is a full-duplex protocol so servers can transmit and receive simultaneously. UDP includes no reliability checking but in the controlled environment of these tests, it is error-free as the ~50,000 synapses/s represent less than 1% of the network capacity.

Discussion

The result of this test indicates that any number of servers can be added to simulate any desired size of neuron array. In practice, other factors will likely emerge with larger numbers of servers and further experimentation will be needed to identify these. Overall, performance remains constant for two or more servers because each server adds the computational and transmission capacity needed to process its neurons and the amount of server-to-server network traffic is constant between any pair of adjacent servers. This also ignores the concept of long connections which will be discussed in the next section.

As it stands, the network transfer implementation is far from optimal even in terms of today's hardware. Here are some additions which could make it significantly faster:

- Use a 10 Gbps network. Estimated performance improvement: 10x.
- Create "virtual axons". Rather than sending individual active synapse weights, the output of a neuron can be transferred to

the receiving server where it is distributed to multiple target neurons. Only a single number (5 bytes) representing the axon must be transferred as all the weight information will reside on the target server. The estimated performance improvement is equal to the simulated number of synapses per neuron. (A side-effect of this change is that learning can be implemented with the synapse data needed residing on individual servers rather than ever crossing server boundaries.)

- Overlap the transmission phase in parallel with neuron processing. This introduces a one-cycle delay in signals crossing machine boundaries which could be an issue. Estimated performance improvement: can reduce the network delay to near zero as neural processing will be slower than network transfer.

Simulating the Entire Neocortex

Based on the performance testing above, we can create an improved estimate of the amount of computer power needed to emulate the neocortex's 16 billion neurons, assuming the improvements above are implemented. Conceptually, each hemisphere could be subdivided radially across *N*.

Short Connections: The number of axons crossing each radial boundary is independent of *N* and is estimated at 50 million. (250 mm * 250 neurons/mm * 800 boundary synapses/neuron). With an expected activity rate of once every 6 s, the expected data load would be 42 MB/s (5 bytes/axon * 50 million axons / 6 s) which is well within the expected performance of a 10 Gbps network.

Long connections: Axons that connect one hemisphere to the other or elsewhere and represent as many as 300 million fibers. We assume that these connections will always cross a machine boundary and must be added to any short-connection calculations. We further assume that they will be distributed evenly among the various machines, meaning that each machine would be burdened with an additional 300M/*N* connections. Regardless of the activity rate, these turn out to be inconsequential relative to the boundary axons.

Using the experimental data, a server simulating 100 million neurons with 100 synapses each can run in one-sixth real time. You

would require 160 such servers to simulate 16 billion neurons, 80 for each hemisphere. Each server would be responsible for transferring 50 million short connections and 2 million long connections. Continuing to use a firing rate of every 6 s and an axon number of 8 bytes yields a data transmission requirement of ~80 MB/s.

Using a different number of synapses per neuron or average firing frequency scales the problem linearly. That is, using 10x as many synapses will make the simulation run 10x slower so one second of "thinking" would require one minute of simulation. Increasing the number of servers will only compensate up to the point where sectors become so small that a short connection will span more than the adjacent sector, dramatically increasing the number of boundary connections.

These performance experiments indicate that creating a full-neocortex simulation is feasible on today's hardware with the scale of the implementation based on various assumptions and the outcome of future neuroscience discoveries. Chief among these is an improved understanding of the actual synaptic interconnection patterns and processes among neurons.

References

[1] L. Abbott, "Lalique's introduction of the integrate-and-fire model neuron (1907)," Brain Research Bulletin, Vol. 50, Nos. 5-6, pp. 303–304, 1999

[2] L. Camuñas-Mesa, B. Linares-Barranco, T. Serrano-Gotarredona, "Neuromorphic spiking neural networks and their memristor-CMOS hardware implementations," MDPB Materials, August 2019. DOI: 10.3390/ma12172745

[3] S. Dutta, V. Kumar, A. Shukla, N. Mohapatra, U. Ganguly, "Leaky integrate and fire neuron by charge-discharge dynamics in floating-body MOSFET," Scientific Reports, 2017.DOI: 10.1038/s41598-017-07418-y

[4] A. Faisal, L. Selen, D. Wolpert, "Noise in the nervous system," National Review of Neuroscience. 2008 Apr; 9(4): 292–303. DOI: 10.1038/nrn2258

[5] K. Grace, ed, "Neuron firing rates in humans," [Survey of related research], AI Impacts, https://aiimpacts.org/rate-of-neuron-firing/

[6] E. Kandel, J. Schwartz, T.M Jessel, Principles of Neural Science (3rd ed.). Elsevier. ISBN 978-0444015624.

[7] P. Lennie, "The cost of cortical computing," Current Biology, March 2003 DOI: 10.1016/s0960-9822(03)00135-0

[8] M. D. McDonnell, K. Boahen, A. Ijspeert and T. J. Sejnowski, "Engineering intelligent electronic systems based on computational neuroscience," in Proceedings of the IEEE, vol. 102, no. 5, pp. 646-651, May 2014, DOI: 10.1109/JPROC.2014.2314776.

[9] J.M.. Montgomery, D.V. Madison. "Discrete synaptic states define a major mechanism of synapse plasticity." Trends in Neuroscience, Dec. 2004, 27(12):744-750. DOI:10.1016/j.tins.2004.10.006

[10] C. Simon, "New Brain Simulator II Open-Source Software" Proceedings, AGI20, in press.

[11] Z. Zeldenrust, W. Wadman, B. Englitz1, "Neural coding with bursts—current state and future perspectives," Frontiers in Computational Neuroscience, 2018, DOI: 10.3389/fncom.2018.00048

[12] Braitenberg, V, Schüz, A., Cortex: Statistics and Geometry of Neuronal Connectiviey, Springer, 1998.

[13] Gerstner W.,Naud, R., Kistler W., Paninski L., Neuronal Dynamics, Cambidge University Press., 2014.

[14] Izhikovich, E., "Simple Model of Spiking Neurons", IEEE Transactions on Neural Networks, Nov, 2003.

Video Links

"Brain Simulator II Tops 2.5 Billion synapses per second"
http://futureai.guru/videos?id=129

"Multiserver Functions with Brain Simulator II"
http://futureai.guru/videos?id=131

"How Your Brain Works Part3: Computational Capacity"
https://futureai.guru/videos?ID=106

Chapter 14:
Future Development

The *Brain Simulator* is an ongoing project with the target of creating an Artificial General Intelligence. The prototype AGI, "Sallie", can do lots of things but, in general, cannot do many of them at once or do them on complex data.

Here are some of the things which Sallie can do right now:

- Move around within a simulator and build up a mental model of surroundings from vision.
- Update the mental model by touch.
- Avoid obstacles while moving in the environment.
- Move objects in the environment to achieve a goal.
- Learn words associated with object features.
- Respond to voice commands and produce spoken responses.
- Imagine the environment from a different point of view.
- Plan a series of actions to achieve a goal.

Development has been on a small scale, limiting Sallie to encountering just a few object types and a few attributes and learning just a few words. The reason for the small-scale approach is the presumption that if we can't solve a problem with just a few parameters, solving it with thousands is beyond impossible. The development intent is to build a system that can truly understand just a few object types before moving on. For understanding, think of how a three-year-old knows about things in her environment. What is there to understand about simple blocks…shape, stacking, falling, inertia, color, planning, goals, following verbal directions, giving verbal descriptions…all things we might associate with a true AGI but on a tiny scale.

With just a few parameters, we can take software shortcuts and learn which processes work and which don't. Once small-scale issues are overcome, the structure of the *Brain Simulator* can be scaled up to huge arrays of neurons or a limitless UKS.

Current development provides for pre-defined object and parameter types. In the UKS chapter, I described how Blue and Brown are both Colors. But how can we make this generalization? We can imagine that somewhere in the brain, every input is just a neuron firing. Now, the difficulty is to determine that some firings represent a shape and others, a color, or a size (or a sound or touch). The key is to know that certain groups of neurons represent a category (such as shape or color). Armed with the ability to infer categories from what would otherwise be seemingly random incoming neural spikes, it is likely that greater intelligence will emerge. Other components such as internal modeling and the ability to learn from mistakes are already in place. The ability to relate words to other inputs is also already in place.

In coming development iterations, Sallie should be able to explore her simulated environment and "understand" what there is to be learned. I put the word understand in quotes because: How much can you learn about a few two-dimensional objects in a two-dimensional world? She should be able to learn that some objects are moveable and that she can move objects to accomplish her goals.

Once the current simulated environment is mastered, Sallie can be upgraded to a three-dimensional simulator. Again, with just a few possible objects and actions, Sallie should be able to learn everything possible about that environment as well. She should be able to learn about object persistence and the passage of time, and planning for the future, and the simple physics of gravity. With these abilities common to any three-year-old, she should be able to expand her horizons to real-world interactions.

Advances will be gradual and, at each step along the way, we'll be able to ascertain that Sallie is safe and progressing toward becoming a useful asset to humanity.

Glossary

This glossary is intended to clarify how terms are used within this book. Also, note that terms (like "Network") are capitalized within the text when they refer to specific Brain Simulator features.

AGI (Artificial General Intelligence): Possible future extension of AI to enable it to perform virtually any mental task a human can.

AI (Artificial Intelligence): Branch of computer science involved in developing systems to perform tasks normally requiring human intelligence.

Algorithm: A procedure or set of instructions that can be followed explicitly to solve a problem. A computer program that can be executed by a CPU is an implementation of an algorithm.

ANN (Artificial Neural Network): A computer system, usually software, designed to loosely follow the computational processes of the human brain involving a large number of identical computing cells.

Axon: The part of a biological neuron that carries the signal from the cell body to the synapses.

Backpropagation: An ANN algorithm for adjusting synapse weights in a neural network that creates learning using the difference between a network's output and a known desired output.

Cache memory: Portion of a CPU that maintains a copy of a portion of RAM content so the CPU can access it more quickly than via a full RAM access.

CPU (Central Processing Unit): Part of a computer that retrieves program instructions from RAM and follows the program to manipulate data.

Deep learning: A neural network with many internal, "hidden", layers.

Dendrite: The part of a biological neuron that receives neural pulses from other neurons via synapses.

Dialog: A custom display window for a Module.

DNA (Deoxyribonucleic acid): The long-chain molecule consisting of a "ladder" of different base pairs which code for the creation of proteins in living cells. DNA can be thought of as a data storage device.

DRAM (Dynamic Random Access Memory): Type of RAM common in computers characterized by the requirement that it must be periodically refreshed or its memory content will be lost.

Graph: An abstract construct of "nodes" connected by "edges" used for knowledge representation.

Knowledge Graph: See "Graph".

LAN (Local Area Network): High-speed connection between computers which is differentiated from Network which is a collection of Neurons connected by Synapses.

Learning: Adjusting synapse weights to allow a network to adapt to perform a specific action, such as learning to recognize phonemes.

Link: Within the UKS, the connection between two Things. It may optionally be weighted or tagged to be processed sequentially.

Module: The software that can be applied to a cluster of neurons to create some unique functionality.

ms (millisecond): A thousandth of a second.

µs (microsecond): A millionth of a second.

Network: Collection of Neurons connected by Synapses (along with Modules and other information). This is differentiated from LAN (a computer network) and neural network (which is a specific kind of AI).

ns (nanosecond): A billionth of a second. Light can travel a distance of about 30 cm in this time.

Neural network: See ANN.

Neural Spike: In biology, the measured voltage spike that travels down a neuron's axon to target synapses. In the *Brain*

Simulator, a spike is the execution of the algorithm which processes a Neuron's Synapses and adds their weights to target Neurons.

Neuron: A biological cell that is a component of the brain and nervous system. Neurons process pulses received from other neurons and transmits pulses to other neurons. Within the Brain Simulator, Neuron refers to a specific simulated entity.

Neuron Engine The portion of the *Brain Simulator* that actually handles the simulation algorithms. This is a separate DLL file from the user interface.

Neuron Server: A stand-alone configuration of the Neuron Engine which receives and transmits all its data via a LAN.

Neurotransmitter: A biological molecule that carries a neural signal across a synaptic gap from one neuron to another.

Phoneme: Any audible unit of speech. A single syllable is usually made up of multiple phonemes. "Ball" is made up of three phonemes consisting of the sounds of the "b", "ah", and "l".

Spike: See "Neural Spike".

Synapse: The part of a biological neuron that transfers a neural signal from one neuron to another using neurotransmitters. Within the *Brain Simulator*, a Synapse is a weighted connection between a pair of Neurons: a source Neuron and a target Neuron.

Thing: Within the UKS, a Thing is a node connected to other Things by Links.

UKS (Universal Knowledge Store): A software implementation of a knowledge graph in a Module

Transistor: An electronic switch with three connections where electricity applied to one of the connections controls the flow of electricity between the other two.

XML: A standard file format that can be used to represent virtually any data. The content of the file more-or-less defines the structure of the data.

Index

N

P

R

S

T

U

W

Z

About the Author

Charles J. Simon, BSEE, MSCS, a uniquely qualified, nationally recognized computer software/hardware expert and neural network pioneer, is also a successful author and speaker.

His combined development experience in CPUs, neurological test equipment, and artificial intelligence software enabled him to write this book.

Previous publications include the book *Will Computers Revolt? Preparing for the Future of Artificial Intelligence,* a book on Computer Aided Design, and numerous technical articles and book contributions, with write-ups in *Newsweek* and other media.

Personal interests include sailing, being one of the few to captain a North American Continent Circumnavigation via the Arctic Northwest Passage and a World Circumnavigation. His philanthropic interests include science centers, art museums, and sailing education programs. Charles and his wife Cathy now split their time between the US East and West Coasts.

Charles is a member of: IEEE, Triple Nine Society, Intertel, Mensa, Ocean Cruising Club, and Annapolis Yacht Club. Charles was nominated for a Microsoft Fellow award.