Polhemsskolan

NeuroCorrelation

Simulations of spiking neural nets

Axel Wickman High School Work

100 points

Class TeInf3a
Technologyprogram
Academic year 2016/2017
Supervisor: Mikael Bondestam

SCHOOL ESSAY, MACHINE TRANSLATED FROM SWEDISH

Abstract

The power of machine learning systems is becoming increasingly clear, as recent advancements in the field has shown. However one problem seems to endure: the slow learning rate of these systems, which is greatly surpassed by that of humans. One proposed solution to this problem is to give the systems general prior knowledge, which can then be used to make assumptions about the dynamics of new environments, which in turn can then be used to solve new problems at an accelerated rate. To do this, a system which can find and model the correlation between external inputs, is proposed. Here a digital simulation of human nerve cells has been constructed in order to model and exploit the fundamental learning mechanics of the brain. Results show that the model behaves like real test show on a cellular level, and that in a bigger network a correlation is indeed found between a small number of given inputs.

SCHOOL ESSAY, MACHINE TRANSLATED FROM SWEDISH

Content

SCHOOL ESSAY, MACHINE TRANSLATED FROM SWEDISH

1. Introduction

In recent years, several major advances have been made in the scientific field of machine learning. Increased computing power, more investment money from companies, and a greater public interest in the area, have contributed to this. There is a great deal of enthusiasm behind the possibility of making computers think more like humans, as it makes it possible to eliminate human shortcomings from certain tasks. A thinking computer can work around the clock, and is often cheaper, safer, and faster than a thinking person. Already today we see machine learning algorithms in full use. On the Internet, they are used to deliver suitable search results and ads, as well as recommend movies, products and songs to the user. This is done by the algorithm producing results with respect to certain information, and then changing depending on how well these results perform. Performance can be measured in different ways depending on the scope of application. If you have a music service that recommends songs, the algorithm can take into[1] account, for example, the user's listening patterns, age, time of day and current weather. These variables then provide a number of song suggestions, via the algorithm. Performance can be measured by, for example, how long the user chose to listen to the recommended songs. In this case, the algorithm would evolve toward recommending songs that users listen to for a long time.

Some other uses for machine learning include diagnosis of diseases, computer vision, speech recognition, language comprehension, robotic motion, control systems and Artificial Intelligence (AI) sometimes another game. However, the algorithms behind these areas vary drastically. Artificial neuron networks (ANN) are used in many cases because of their ability to find complex relationships. They consist of neurons that can connect with each other with electrical signals via couplings. Each connection goes from one neuron (the presynaptic cell) to another (the postsynaptic cell), and has a certain strength that affects the amplitude of the signal impulses that pass through. When a neuron receives signals through connections, they are processed and passed on. Learning then occurs by varying, for example, the strength of the couplings, or vary how each neuron treats the signals. The more neurons and connections a network has whose values are not determined by a human being, the "deeper" it is. There are many types of neural networks. Among other things, thestrusion of the networks may vary. In the most common type, the neurons are organized in layers, and the signals can only move forward. This is a feed-forward network, in contrast to the reconnecting network. You can have different types of signals in the networks. Often it is just a simple number value, but there are exceptions as in convolution neuron networks (CNN), which have proven to be very good at, among other things, computer vision. These networks are inspired by the visual cortex of our own brains (although most studies were done on cats), and have been a major source of development in machine learning in recent years.

A source of inspiration for my project was the publication "Building Machines That Learn and Think Like People". Here the problem is discussed as to why we humans can learn to perform tasks such as playing a game in just a number of attempts, while it can take hundreds of hours for an artificial neuron network to get close to our performance. The authors' view is that it is due to fundamental differences between the artificial networks used and developed today, and how we humans work. [2]They suggest that we humans use the knowledge of the world we already have, and apply it in the new context. This allows us to make assumptions that the artificial network needs to test itself up to. Usually when you want to train a network, you start with that it is completely untrained, with no

[1] Adam Spector. Spotify Just Dove Deep Into Machine Learning Personalization. LiftIgniter. 28 Maj 2015. http://www.liftigniter.com/spotify-just-dove-deep-into-machine-learning-personalization/ (Retrieved 2016-10-30).

[2] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, Samuel J. Gershman. Building Machines That Learn and Think Like People. CBMM. 2016. arXiv:1604.00289v2

experience at all. By first allowing a neuron network to build up intuitive models such as physics and psychology, it gives it a basis on which to base its development on when it has a new problem to solve, according to the authors. This is called model-based learning because the network builds a solution based on its model of problem environment behavior.

Building a kind of network system that leverages these theories of model-based learning, and then training it up, I think would be a big step towards general AI – where every trained network (brain) is used for more than just solving single problems. A first step would be to create a network that automatically builds an intuitive model of the values you give it. Intuitively, there is no reasoning in the network regarding the connections it has learned, which is a consequence of the fact that only basic learning mechanisms will be used. Getting the network to steer towards solving problems according to how well I as a programmer value its performance is not included in this project but is a possible future researcharea.

1.1. Purpose and Question

The purpose of this project is to find out how to build up a computer simulation of a biological brain, which can then be used for machine learning. The question is how to build this system, and whether a biological model works to build an internal intuitive model of the relationships between input values.

1.2. Materiel and method

1.2.1. Nätverksmodell

The human brain is a good example of the kind of system I want to build. You can think of it as a correlation machine. It takes inputs in the form of nerve signals from our minds, and finds connections between these. These relationships are developed through relatively simple rules that work at the cellular level, and mean that the connections between neurons, where there is a correlation in the firing patterns,, become stronger. The rules were theorized by canadian researcher Donald Hebb in 1949 in his, in neuropsychology, very famous book "The Organization of*Behaviour*". " Hebbian" learning models are very common in artificial neuron networks, although their implementation of necessity varies depending on network type. The Hebbian learning model only affects the strength of the connections between the neurons, and can be summarized: Cells that are fired together are interconnected. If one nerve cell repeatedly causes another to fire, the connection between them is strengthened. This is called Long-term Potentiation (LTP). The opposite of this is called Long-term Depression (LTD). This ability to alter the strength of the couplings is called synaptic plasticity. [3][4]

There are two popular ways to represent biological neurons in network models. For example, you can represent the firing rate of nerve cells, a so-called rate model. This model is robust for learning, easy to implement, and is often biologically representative, but it is unable to represent all kinds of information transfer. Neural coding is how information can be represented and moved in the brain. A rate model allows information to be represented only in how active the neurons are. Since the purpose of this project is to create a relatively realistic biological model, it is necessary to allow temporal coding. This is by instead representing the electrical voltage potential, in both the neurons and the couplings (which are henceforth referred to as synapses). Over time, the tension can change. If the voltage exceeds a certain threshold potential, it causes a nerve impulse, which is passed on to synapses and thus other neurons. This makes the model an impulse-based network (SNN). This

[3] Hebb O. Donald. The Organization of Behaviour. New York: Wiley & Sons. 1949.

[4] Hebb's Three Postulates: from Brain to Soma. [online video]. 2015. https://www.youtube.com/watch?v=SIp_CTEfiR4 (Retrieved 2016-10-30)

temporal encoding means that a single neuron can transmit information via specific firing sequences, which in practice gives it the technical ability to communicate with, for example, Morse code. Using an SNN also allows to use a more realistic model for synaptic plasticity. Impulse-time-plasticity (STDP) bases the change in synapse's strength on the relative time between nerve impulses. This model is in full compliance with Hebbs rules, and has proven to represent the biological reality quitewell.[5]

As mentioned earlier, neuron networks can consist of different kinds of structures. In this project, the neurons are spatially placed in 3D space, creating connections to all the other closest neurons. The coordinates are randomly generated decimal numbers, which are evenly distributed within a sphere. This system means that the brain has a reconnecting neuron structure. Here the problem arises with positive feedback loops, when signals in the brain can feed themselves for an unlimited amount of time. This can happen in even reality under certain genetic conditions, and is known as epilepsy. One of hjärnans way of counteracting this is by placing inhibitory synapses, which inhibit activity in neurons. This type of synapses are usually more localized, have shorter connections than excitatory synapses (which promote activity in neurons). They are also rarer, as only 15% of neurons in the brain beam have an inhibitory role. Their role in actual learning and information processing is controversial in the scientific community, but it is clear that they have an important role in countering epileptic seizures..[6]

Since the brain has no mechanism that "motivates" it to pursue certain goals, it is pointless to build a test that measures brain performance. If this were possible, you could train the brain, and then rewarded predictability, which would be extracted as nerve signals in certain areas. However, this solution would have had the disadvantage of not knowing how much of the predictive capacity that arises as due to the natural (STDP) tendency of the synapse to find correlations, and how much is just a product of the reward system. For this reason, I therefore choose to measure brain learning through direct analysis of the network. A rendering engine is very helpful here as it becomes possible to quickly and easily make an intuitive assessment of the behavior and ability of the brain, which is beneficial during the development and design of the simulation system. Through other tools, it is then possible to make a more objective and thoroughassessment of the system. For example, it is possible to observe which areas are active when specific inputs are provided.

1.2.2. Programming

The programming language selected sets the basics of the structure of the application, how detailed the code needs to be, the project workflow, where the application can run, and how quickly the code runs. Since the code in this context would be used in a simulation that over time evolves towards a certain state, which is then analyzed, it is advantageous that this condition is reached as quickly as possible in relation to real time. Therefore, a quick language was required, which usually entails a low-level language, where the programmerare has great power and responsibility to define the algorithms in detail by working close to the hardware. Best suited for this is C++, which also has the advantage that it works well in 3D environments. C++ is a further development in the C language, and has more built-in features while providing a lot of control. The language supports object-oriented programming, which makes it suitable for larger simulation projects, where you have many different types of objects that all have shared behavior.

Because C++ is a language that needs to be compiled before it runs, you need to select a compiler. The role of the compiler is to translate the programmer's c++ code into the machine code running on the processor. In this project, uses MingW compiler on Windows. This is becausethe compiler is well

[5] H. Markram, W. Gerstner, P. J. Sjöström. Spike-Timing-Dependent Plasticity: A Comprehensive Overview. Neurosci Synaptic Front. 2012. doi: 10.3389/fnsyn.2012.00002

[6] CCNLab, CCNBook/Networks, https://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Networks, 31 March 2015, (Retrieved 2016-11-20)

associated with the CodeBlocks programmingenvironment. When installing CodeBlocks can be selected to also install MingW. To handle the 3D graphics in the renderer, an API (application programming interface) is required that can speak to the hardware, and can work in a wide range of software environments. Here I choose OpenGL, which is a kind of standard that hardware manufacturers have implemented in their devices. This standard allows the same code to run and operate (approximately) equally on different systems. OpenGL responsibility, however, only for rendering pixels on a screen surface, it requires a different API to talk to the operating system in order to create a window. Here I use GLFW, which is cross-platform (works on multiple operating systems), and can manage and forward user input to the rest of the application.

2. Usage and results

2.1. Building a programming environment

To manage all files during the project, GitHub (project link is available under attachments). This makes it possible to control the change history, share my code, and have updated backups on a server. A number of external references were also required, so extra work was required to configure the programming environment. After CodeBlocks and MingW had been installed, I downloaded and installed these external references, which are so-called bibliotek that contains code that other projects can import and use.

Table 1 Shows the nameofthe library used in the code, as well as the function they play.

Library	**Use**
GLFW (& OpenGL)	Create windows and render 3D graphics.
GLM	Store and process arrays and vectors
GLEW	Management of OpenGL add-ons
PicoPNG	To load images from .png format
dear imgui	User interface programming
Boost	Stable vector container
TinyExpr	Interpret and calculate arithmetic formulas

The MingW folder, whose location was selected during the installation of CodeBlocks (default location is in the CodeBlocks folder), has a subfolder named "bin." This folder should be added to the PATH environment variable, as it contains multiple applications that need to be called during library installation. The installation is done by moving the folder with the source files (.h, .cpp, or .hpp) to under "includes" in the MingW folder. The installation of these libraries varies and takes place as follows:

1. PicoPNG, dear imgui, TinyExpr - Already embedded in the project files and follows down when the Git project is downloaded.
2. GLM, Boost - "glm", and "boost" folders are added as subfolders in the "includes" folder in the MingW folder. This allows the compiler to find and use the files when the program is created.
3. GLFW - Downloaded precompiled for Windows 32 bit. In the downloaded ZIP file is a folder named "include" that is merged with it in MingW. The precompiled files are located under "lib-mingw" in the ZIPfile. His, the two .a files should be added under "lib" in MingW. The remaining . The .exe file should be placed in a place known for the project .exe file after compilation, for example, in the same folder as the one, or in mingw's "bin"folder that was previously added to the PATHvariable.
4. GLEW – Available in pre-packaged versions, but these do not work for mingw-32, which means that the compilation needs to be done manually. First, the source files should be downloaded (available on the website), and extracted to their own folder. Then the command prompt can be opened in the folder (Ctrl + right-hand,and the following commands can be executed::
    1. gcc -DGLEW_NO_GLU -O2 -Wall -W -Iinclude -DGLEW_BUILD -o src/glew.o -c src/glew.c
    2. gcc –nostdlib -shared -Wl,-soname,libglew32.dll -Wl,--out-implib,lib/libglew32.dll.a -o lib/glew32.dll src/glew.o -L/mingw/lib -lglu32 -lopengl32 -lgdi32 -luser32 -lkernel32
    3. ar cr lib/libglew32.a src/glew.o

This creates three files in glew's "lib" folder with two .a- and one . DLL file that can be installed in the same way as those in GLFW. There is also a "includes" folder that should be merged with it in MingW.

2.2. Structure and basic systems

The structure of the program is designed in such a way that the simulation object (the brain itself) can be independent from the renderer, which is given only the memory address of the simulation. The renderer can thus extract information that is then reprocessed to pixel values that are then sent to the screen. The renderer can also change the simulation via this memory address. This allows the user to change, for example, simulation speed during driving time. The simulation object stores different types of simulators. Common to these simulators is that they all know the simulation object, and that they have a driving function that updates the object in relation to the simulation time variable. The fire simulator makes sure to fire certain neurons according to a certain frequency defined in the main function. It is this system used to give the brain the input values that are then processed.

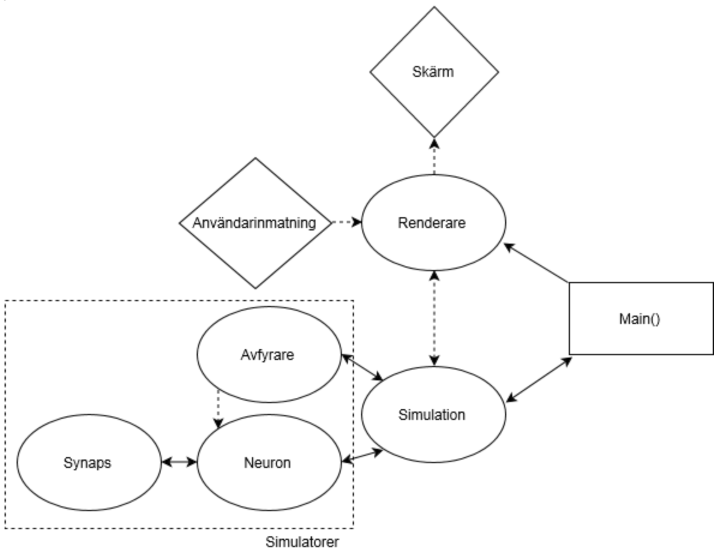


Figure 1 Shows the functional relationships between the different classes (ovals), user interface (lozenges), and the main function (rectangle). Dashed line indicates that informationexchangeoccurs. Solid arrows between classes indicate that the class on the right owns the class on the left. The classes within the dotted rectangle inherit all of the simulator class.

The simulation object owns a simulation list whose task it is to run all simulators at the right time in the right order. Each simulator is responsible for scheduling opportunities to run in the future in this global list. Each opportunity is automatically sorted so that what is closest in time is first. After an opportunity has passed and the simulator has run, the element is deleted from the list. In practice, this system means that only what is necessary to simulate needs to be simulated, and so that the simulation speed can vary without changing the behaviour of the simulation.

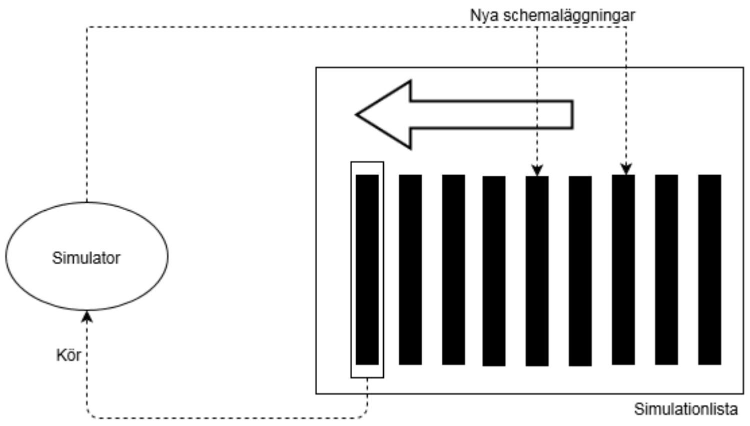


Figure 2 Shows how the simulation list calls the driving function of the simulator to which the closest in time (far left) time. The simulator may then schedule later simulation sessions in the future, which will then be sorted in depending on the time in the simulation list. After a simulation session has been called, it is deleted, and the next opportunity will be first in the list.

By developing both the simulation and the renderer in parallel, it will be possible to more easily sort out bugs, and understand how the system behaves under different conditions. In C++, it is common for classes and global variables to be first declared in header files (.h), and then defined in the code files (.cpp) themselves. This contributes to a better overview of the project, and makes it easier to use the classes in other contexts without getting in touch with the internal code that the class itself uses. In this project there are mainly two important header files. "NeuCor.h" declares the following classes:

```
// Själva simulationen av hjärnan
class NeuCor {

// En schemaläggning med planerad tidpunkt och minnesadress till simulatorn
struct simulation {

// Abstrakt klass som ligger till grund till allting som simuleras
class simulator {

// Avfyrar neuronerna efter viss given frekvens, och är en typ av simulator
struct InputFirer: public simulator {

// Neuron-klassen är en typ av simulator
class Neuron: public simulator {

// Synaps-klassen är en typ av simulator
class Synapse: public simulator {

// Enkel datatyp som förvarar 3D koordinater som float-värden (decimaltal)
struct coord3 {
```

Snippet 1 Shows which classes and structures NeuCor.h declares.

"NeuCor_Renderer.h" declares:

```
// Renderaren, som skapar ett fönster och ritar ut en visualisering av hjärnan i
3D
class NeuCor_Renderer {
```

Snippet 2 Shows which classes and structures NeuCor_Renderer.h. declare.

By dividing the simulation classes and the renderer by themselves, it will be possible to only include the simulation in your project, if only it is necessary. This means that in this case you do not need to install all the libraries that the renderer needs.

2.3. Implementation

When C++ creates an object of a particular class, a constructor is called, a variety of initialization function whose task it is to construct the object. The variables defined in the header file already exist in the new object, but they do not necessarily have any assigned values. In my project, this means that the brain is empty when it is first created, and that in the art ucator (or later) neurons and synapses are needed. I do this by giving the brain (NeuCor class) constructor an integer as an in parameter, which specifies how many neurons (with partially random properties) should initially be deployed. This is done in two stages: first all the neurons are created and stored, then the connections (synapsare) are created. The connections are created by each neuron measuring the distance to the other neurons, and if this distance is less than 1 unit length, and the link does not already exist, a synapse is created that goes from the presynaptic neuron to the postsynaptic neuron. This synapse is stored in a container inside the presynaptic neuron. This method means that there will always be two connections in each direction when two neurons aresufficiently close toeach other. However, thealgaem is ineffective as it has to compare the distances to each neuron for each neuron, creating an exponential growth in the number of calculations that must be performed as the neuron count increases (a so-called O(n^2) – algorithm within Big O notation). This problem can be solved in the future by placing the coordinates of each neuron in a kd tree,a binary partitioning container that reduces the number of necessary comparisons linearly with the number of neurons. However, this is outside the scope of this project. When the neuron and synaptic objects are created, their constructors, which in both cases are responsible for defining the properties of the objects, such as: resting potential, switching strength, form of impulse voltage over time, etc. are also called. Synapse artificial scans also put an identifier of itself in a special container in the postsynaptic neuron, which allows this neuron to know what input synapses it has, even if it does not own them.

Neurons can be created indirectly in the brain outside the art u.s. by calling one of the functions of the brain object:

```cpp
// Initierar hjärnobjekt med 0 neuroner.
NeuCor brain(0);

// Skapar neuron med koordinaterna: (-2.0, 1.25, 0.0)
brain.createNeuron({-2.0, 1.25, 0.0});

// Skapar neuron med koordinaterna: (0.0, 1.0, 0.2)
brain.createNeuron({0.0, 1.0, 0.2});

// Skapar neuron med koordinaterna: (0.0, 0.0, 5.0)
brain.createNeuron({0.0, 0.0, 5.0});

// Skapar neuron med slumpmässiga koordinater
brain.createNeuron({NAN, NAN, NAN});

// Skapar kopplingar mellan alla närliggande neuroner
brain.makeConnections();
```
Snipe 3 Shows how it is possible to create neurons in the main function through the brain object.

When NAN (not-a-number) is specified, the coordinates are created randomly, evenly distributed within a sphere. This is done by generating 3 random real numbers (x, y, z) within a certain range. If these numbers, when interpreted as 3D coordinates, are longer than a certain distance from origin, the numbers are rejected, and three new ones are generated, otherwise the numbers are used as the coordinates of the neuron. This algorithm is easy to implement, but relatively inefficient as it rejects some of the coordinates. The probability that the coordinates do not need to be rejected can be calculated by:

$$Volym_{Sfär}\Big/Volym_{kub} = \frac{4}{3}\pi r^3\Big/(2r)^3 = \frac{\pi}{6} \approx 52\%$$

There r is the maximum distance from origin.

After this, the brain is initiated, and ready to be simulated. It is here necessary to slice the code to the threesimulators, as well as the brain object. In the brain's global driving function, background activity is implemented by giving each neuron får a chance $1/600$to fire spontaneously. The driving function of the neuron is regularly called by, among other things, the brain object (through the simulation list). This function updates the condition of the neuron depending on how much time has passed since it was last called.

```
void Neuron::run(){
    // Hittar nuvarande tid och tidsskillnad
    float currentT = parentNet->getTime();
    float deltaT = currentT - lastRan;
    lastRan = currentT;

    // Går ur funktion om ingen tid har passerat (för att undvika oändliga loopar)
    if (deltaT == 0) return;

    // Integrerar potentialen av aktiva insynapser
    charge_insynapses(deltaT, currentT);

    // Växer / avtar neuronens potential exponentiellt mot basnivån
    charge_passive(deltaT, currentT);

    // Kontrollerar om neuronens potential är över tröskeln,
    // och avfyrar i sådana fall neuronen
    charge_thresholdCheck(deltaT, currentT);

    // Styr potentialen efter funktion om neuronen håller på att avfyra
    AP(currentT);

    // Uppdaterar aktivitets-variabeln
    setActivity(firings/(((float) currentT-activityStartTime)/10.0));
}
```

Snippet 4 Shows the appearance of the neuron-class run function.

The activity variable, which keeps track of avfyrning frequency, is also calculated. The four other functions called regulate the current electrical potential of the neuron. The member function (i.e. a function that belongs only to the class) "charge_insynapses" is the neuron's connection with its in-synapses. This function increases neuron potentialen one with: where a is the number of active $\sum_{i=0}^{a}(f_i * \Delta t * g(\Delta t))$synapses,Δt is time since last update, f_i is the impulse strength of a specific synapse,and is a function developed by $g(\Delta t) = 0.9943 * e^{0.3702*\Delta t}$regression that makes the behavior the same regardless. Δt A synaptic shrimp as active in 2 ms from the beginning of an impulse. This function is a greatly simplified model of how dendrites absorb neurotransmitters in the synaptic gap, and how these signals are then summed up in the neuron. Instead ofintegrating an impulse voltage change over time, only a constant value is integrated over time. This simplifies the mathematical calculations that the computer needs to perform and makes brain behavior less complicated.

After the impulses of the in-synapse have been integrated and summed up, "charge_passive" is called,whose task it is to get the neuron's potential to move towards its resting potential (b), which is -70 mV, over time. This is done exponentially through the function , where c is change per ms, which in the simulation is 0.5.$p_{uppdaterad} = (p_{nuvarande} - b) * c^{\Delta t} + b$

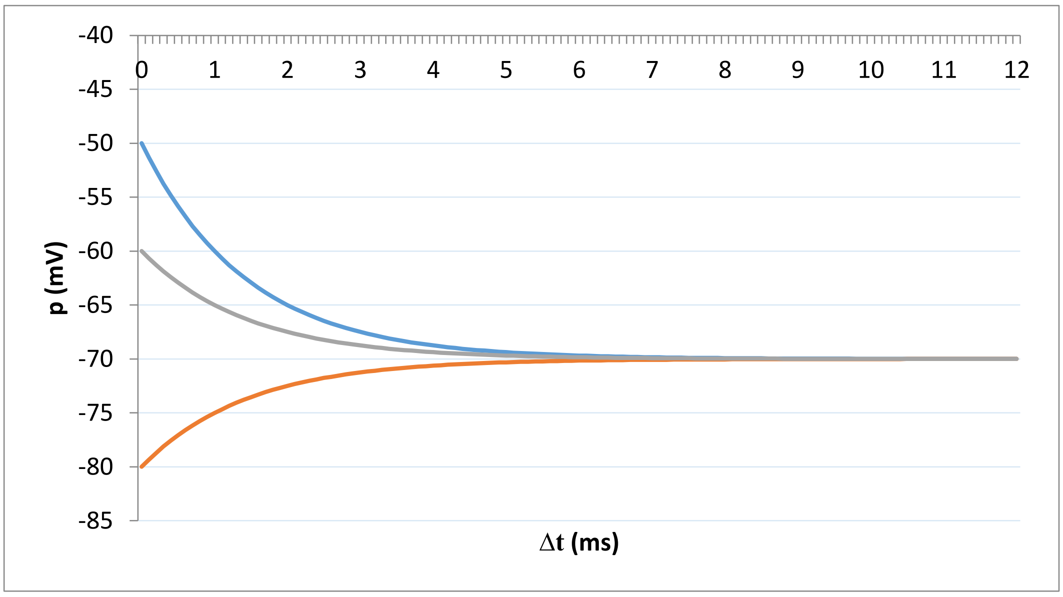


Figure 3 The example shows that the potential (p) always moves towards resting potential over time since last run (Δt) regardless of the starting value.

The next feature, "charge_thresholdCheck" checks whether the neuron's potential is above a certain threshold potential (-55mV). If this is the case and the neuron or synapse is not already being fired, the cell's firing function is called. This function performs a number of tasks. It increases the neuron's launch counter (used to determine activity) by 1, sets the trackvariable to 1, and sets the neuron's variable for last firing to the current time. In addition, it fires the STDP learning algorithm in all in synapses, and calls all out-synapses' firing function using impulse strength as a parameter.

The last feature that changes the potential of the neuron class is "AP". This function gives the shape of the neuron's tension during a nerve impulse. To understand the reason for this form, it is necessary to understand the underlying biochemical process behind it. In this context, the concept of potential is the difference between the electrical voltage inside and outside a neuron's cell membrane. Voltage occurs due to different ion concentrations, which is a result of the sodium-potassium pumps found in the cell membrane. These protein pumps work to maintain sodium ions$^{(Na+)}$outside the cell and potassium ions (K+)$^{+}$insidethe cell. Without these, the ion concentrations outside and inside the cell would be the same because the membrane is super permeable for potassium ions and partially permeable for sodium ions. This is what causes the membrane potential to move towards resting potential (-70 mV). A nerve impulse is a rapid and sharp change in ion concentration, and thus also the electrical potential. This change occurs when voltage-controlled ion channels, which are also present in the membrane, open and close the flow of ions depending on the current voltage in the membrane. When excitatory in-synapses are activated, positively charged ions flow into the neuron, increasing the electrical voltage. If the electrical voltage exceeds a certain threshold potential (-55 mV), the voltage-controlled sodium channels that are normally closed are opened, causing sodium ions to flow into the cell. This causes the membrane potential to rise (to about +30 mV) and is called depolarization. On this occasion, the sodium channels begin to close, and the sodium ions begin to be pumped out of the cell again due to the sodium pumps. At the same time, the otherwise closed voltage-controlled potassium channels begin to open, causing the potassium concentration inside the cell to decrease, and thus also the cell's membrane potential. This is called repolarization. When resting potential is reached, these potassium pumps do not close immediately, but it continues to leak potassium ions out of the cell. This means that the membrane potential is temporarily below the resting potential, which is called hyperpolarization. The potassium pumps then restore the potassium ion concentration, and the neuron has completely restored ion concentrations and a restored membrane potential.[7]

This biological behavior is modeled in the AP functions, but will be represented here with mathematical note. The value given by the V(t) function (where t is the time in ms since the last launch began), becomes the neuron's potential if a launch is ongoing. By simulating the relative amount of sodium and potassium ions over time, and then summing up these two amounts, it is possible to determine how the neuron's potential changes over time. My assessment here has been to use Gaussfunction to model the relative ion amounts during a nerve impulse..

$Na^+(t) = a_1 e^{-\frac{(t-d_1)^2}{2{w_1}^2}}$

$K^+(t) = a_2 e^{-\frac{(t-d_2)^2}{2{w_2}^2}}$

Here, too, is a time since the last launch began. I have chosen to use the values in the simulation:
$a_1 = 1;\ d_1 = 1;\ w_1 = 0.3;\ a_2 = -0.2;\ d_2 = 2.16;\ w_2 = 0.6$
Note that the value a_2 is negative, as the relative amount of K+$^+$ inside the cell membrane first decreases, and then increases.

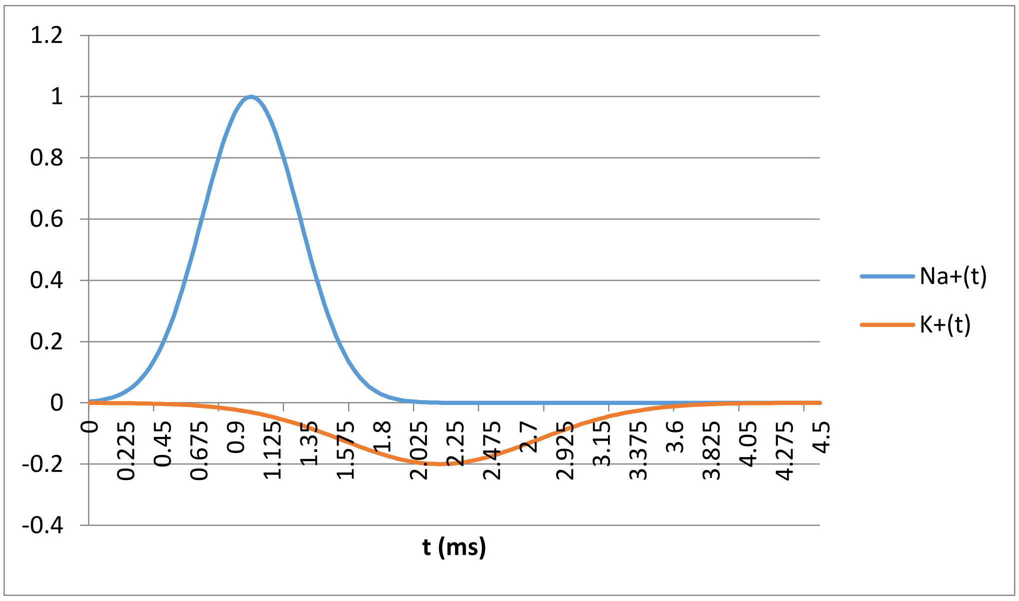


Figure 4 Shows how the relative amount of Na+$^+$and K+$^+$ions changes over time since the beginning ofimpulse (t) relative to each other.

By building on these two functions, v(t) can be constructed:

$$V(t) = f * \left(Na^+(t) + K^+(t)\right) + b + (T - b) * \max\,(1 - t;\ 0)$$

Here is b resting potential (-70 mV), T threshold potential (-55 mV), and f is 100. The last term is added to ensure that impulse potential begins on threshold potential instead of the resting potential. The max function returns the highest value of the input variables specified. Note that the V(t) function only applies for 2 ms from the beginning of the launch. This is because of the assessment I have made that an impulse counts as ongoing in 2 ms (with these selected constant values). After this time, it is possible for the neuron to be fired again.$(T - b) * \max\,(1 - t;\ 0)$

[7] Khanacademy, Neuron action potentials: The creation of a brain signal, https://www.khanacademy.org/test-prep/mcat/organ-systems/neuron-membrane-potentials/a/neuron-action-potentials-the-creation-of-a-brain-signal (Retrieved 2017-02-28)

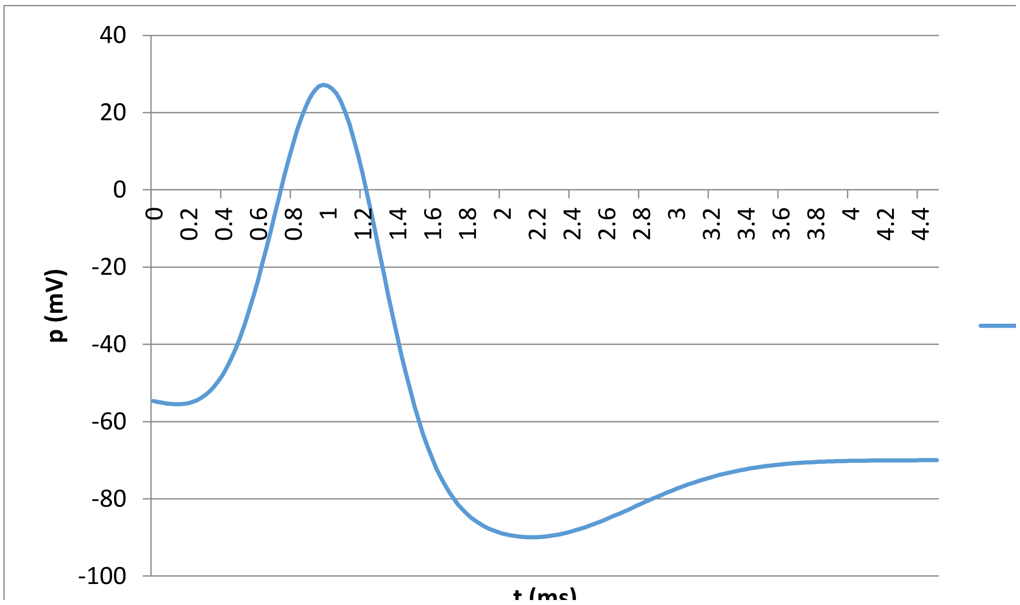


Figure 5 Shows how the potential (p) changes over time since the beginning of impulse (t) when using the V(t) function is used.

When a neuron fires a synapse, which occurs at the beginning of an impulse, the synapse prematurely calculates how long it will take for the impulse to reach the target neuron by multiplying the signal propagation rate of the synapse (2 length units per ms) by the length of the synapse. At this future time, a call is scheduled for the call of the call of the synapse run function. When the run function is called, the following occurs: a run of the target neuron is scheduled at the end time of the impulse (after 2 ms), the synapse stores the current time in the "lastSpikeArrival" member variable, and calls the "synapticPlasticity" plasticity function. This function alters the strength of the synapse, which is a factor in calculating impulse strength. In other words, a strong synapse has a greater impact on the target neuron. An excitatory synapse has easier to pass on an impulse, and a strong inhibitory synapse (with a negative strength value) has easier to prevent the target neuron from being fired. The plasticity function (which implements STDP learning) is called, as previously mentioned, on two occasions: when the target neuron is fired and when a signal reaches the target neuron in the synapse. The function assumes that there is a trace immediately after a firing in all neurons and synapses respectively. This track is set to 1 at launch, and then declines exponentially overtime (65% per ms in neurons, 75% decline per ms in synapses). The calculation of the value of the two track variables is made by: , and , where S are track variables, a is decreasing per ms (a$S_n = a_n{}^{t_{nu}-t_n}S_s = a_s{}^{t_{nu}-t_s}$$_n$ = 0.65, a$_s$ = 0. 75), t$_{\text{is now}}$ the current time, t$_s$ is the time when a signal last reached the target neuron in the synapse ("lastSpikeArrival"), and tn$_{\text{is}}$ time at the beginning of the last impulse in the target neuron. It is important that the track variables are updated after the plasticity function has been called for learning to work as intended. This is ensured by allowing the variable to be 0.0 and it was exactly equal to 1.0. When the track variables are calculated, they are used to calculate the resulting difference in the strength of the synapse through , where f$\Delta w = f_g * (f_s * S_s - f_n * S_n)$$_g$ is the global learning factor (with a default value of 1.0) used to control how rapidly the synapse forces are changing globally. The variables f$_s$ (default value 0.13) and currently$_n$ (default value 0.30) change the plasticity function's preference for negative compared to positive Δw . After applying the difference in synaptic strength, value-clamp is used to keep strength within a certain range (0.0 to 1.0 for excitatory synapses, -1.0 to 0.0 for inhibitory).

2.4. Use

2.4.1. Library

A user of the library imports my code into his project, and can thus use my simulation model without having to understand how the simulation works in detail. An understanding of the external interfaces I have designed is instead sufficient. The code can be cloned with GitHub

(link to project can be found under Appendix 1) . Cloning can be done through standard console-based GitHub, although I recommend using GitHub Desktop as it has a graphical user interface that is easier to use. After this, the user is required to go through the process of installing the required libraries, as specified in section 2.1Building a programming environment below.

```
NeuCor(int n_neurons);
// Constructs the class. n_neuron parameter is the number of neurons created
initially

void run();
// Runs the whole simulation

float runSpeed;
// How much time (in ms) is simulated when run () is called (default value 1.0)
bool runAll;
// True/false value about whether all neurons should be simulated (default value
false)

float getTime() const;
// Returns how much simulation time (in ms) has elapsed

float learningRate;
// Is the global learning factor fg (default value 0.6)

float presynapticTraceDecay, postsynapticTraceDecay;
// Decrease per ms of the track variables of all synapses and neurons (default
value 0.5, and 0.9)

void setInputRateArray(float inputs[], unsigned inputCount, coord3
inputPositions[] = {NULL}, float inputRadius[] = {NULL}));
// The "inputs" parameter is an array of float values that indicate (in Hz) the
firing frequency of the firers present in the brain. "InputCount" indicates the
number of elements in this array. Because it is the memory address of the array
that is stored, the firers are automatically updated according to the values they
have at the memory address when run () is called. The parameters "inputPositions"
and "inputRadius" are not necessary, but define the positions and radius of the
firers in the brain.

void addInputOffset(unsigned inputID, float t);
// Shifts the given firing (inputID) of the given firing by the given time t (ms).
It is also possible to use negative time values.

void createNeuron(coord3 position);
// Creates positions at given coordinates. If NAN is specified as coordinate
values, the position becomes random.

void createSynapse(std::size_t toID, std::size_t fromID, float weight);
// Creates synapse between from one neuron ("toID") to another ("fromID") with
given strength ("weight").

void makeConnections();
// Creates two connections between all neurons less than 1 unit of length apart.
```

Snipe 5 Shows the use and function of the public members of the "NeuCor" class.

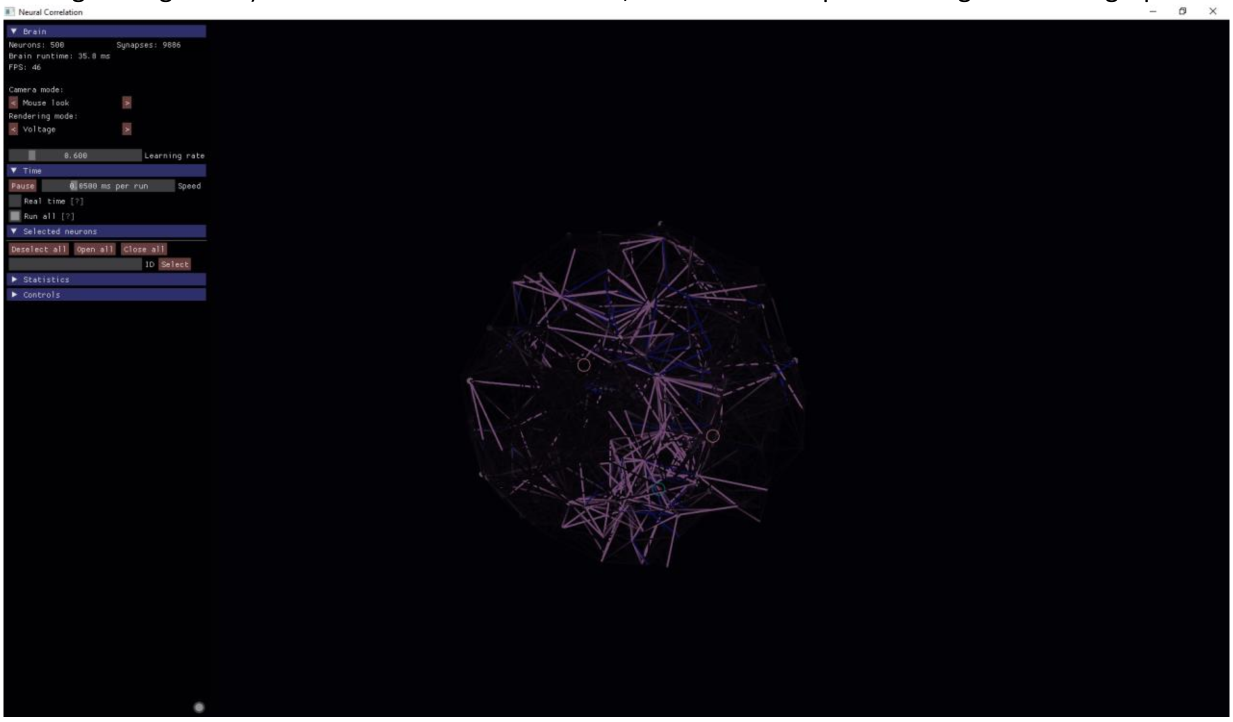
The renderer has more public member variables and functions. However, only the necessary ones will be explained when several of these are either self-explanatory, or are intended to change the status of variables that can otherwise be changed within the user interface by external code. Documentation for these essential members of the "NeuCor_Renderer" class is below For an example program demonstrating how "NeuCor" and "NeuCor_Renderer" can be used together, see Appendix Appendix 3.

```
NeuCor_Renderer(NeuCor* _brain);
// Constructs renderers. The "_brain" parameter is a pointer to the brain.

float getDeltaTime();
// Returns real time that has elapsed since running updateView ()

bool runBrainOnUpdate;
// If the renderer should run the brain in when updateView () (default value
false)

void updateView();
// Renders the brain and displays it in the window

void pollWindow();
// Requests Windows to specify the actions performed by the user. This feature is
necessary for Windows not to detect the window as frozen.
```

Snipe 6 Shows the use and function of public members of the "NeuCor_Renderer" class. Building a programming environmentBuilding a programming environment

2.4.2. Graphical interface

A precompiled version of the code is available for download under Appendix 2. The graphical interface consists of both the rendering of the brain, and the panels that the ImGui library is used to create. With this interface, it is possible to read information about the brain to understand its behavior, and to some extent also to change the way the brain is run. When the renderer first opens the window (see Figure Figure 6you can use and move the menus around. Clicking on the heading "Controls" opens a list that details how to use keyboard and mouse to manipulate the rendering. Measurement will be made with the three graphs located under the menu "Statistics" (see Figure Figure 7, Figure 8, Figure 9(which is shown when a neuron is selected and its window opens, seeFigure Figure 10synapse handlebar graph (located under synapsmenus in the neuron window, see Figure Figure 11). Due to limitations with ImGui, it has not been possible to grade these graphs.



```
Neural Correlation
▼ Brain
Neurons: 500          Synapses: 9886
Brain runtime: 35.8 ms
FPS: 46

Camera mode:
< Mouse look      >
Rendering mode:
< Voltage         >

0.600             Learning rate
▼ Time
Pause  0.0500 ms per run   Speed
Real time [?]
Run all [?]
▼ Selected neurons
Deselect all  Open all  Close all
                  ID  Select
▶ Statistics
▶ Controls
```

Figure 6 Shows an example of how the rendering of panels (left) and the brain looks like on the screen. The synapse appear as streaks, white filled-in rings are neurons, non-filled rings are firemen.

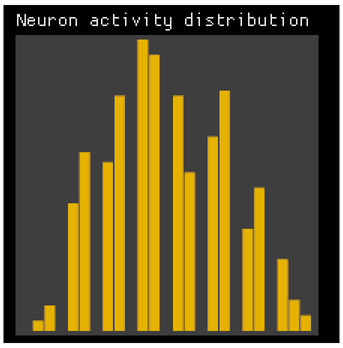


Figure 7 Shows interface graph (under the menu "Statistics") indicating the distribution of firing frequencies (x-axis) of neurons in the brain. The graph's properties can be changed by clicking on it..

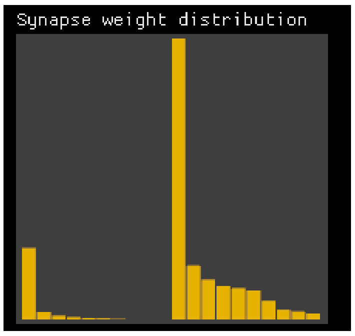


Figure 8 Shows interface graph (under the menu "Statistics") indicating the distribution of firing frequencies (x-axis) of neurons in the brain. The graph's properties can be changed by clicking on the graph.

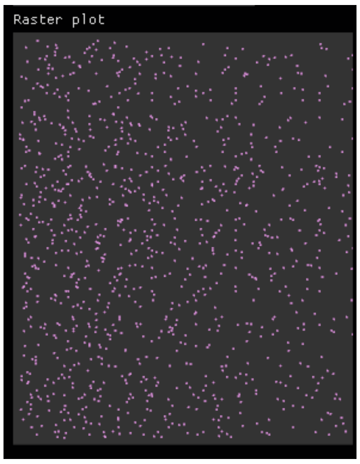


Figure 9 Displays the raster graph (under the "Statistics" menu) of the firing of neurons. Every dot is firing. X-axis is time (closest in time to the left), y-axis is the neuron's ID number. The range of the X-axis can be changed by clicking the graph.

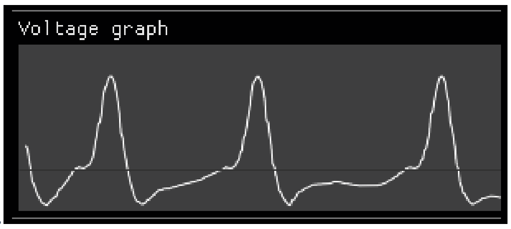


Figure 10 Shows the potential graph (neuron window) of the voltage (y-axis) of the selected neuron over time (x-axis, nasti in time to friend's). The dark line marks threshold potential.

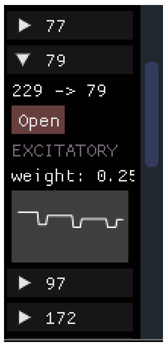


Figure 11 Shows synapse information (synapse list along the bottom of neuron window), for the synapse that goes from the neuron with id 229 to the target neuron with id 79. The graph shows synaptic strength (x-axis, closest in time to the left) over time.

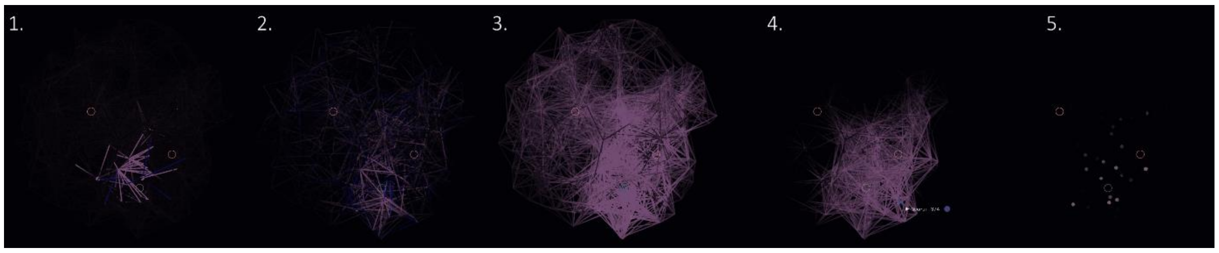


Figure 12 Displays the different rendering modes. Mode 1 shows tension of synapses as well as the type of synapses (pink is excitatory, blue is inhibitory). Mode 2 shows synaptic strength (pink is excitatory, blue is inhibitory) with the firing sequence as a factor (can be turned off in menu) to make the main synapse stand out most clearly. Mode 3 shows the firing rate of the presynaptic/postsynaptic neuron (depending on the end of the synaptic line). Position 4 shows how the signals spread from the marked neurons. This takes into account synaptic strength. Mode 5 renders only neurons.

In addition to these graphs, there are also a number of rendering modes that present information in the rendering by changing the transparency of the synapse. The five different rendering modes are explained in Figure Figure 12. Rendering mode showing launch sequences has extended functionality (Figure Figure 13which is useful for analysis as it illustrates how activated certain parts of the brain are in different states. Changes in activity patterns in the network can either occur because plasticity has changed the way signals are distributed, or because the inputs that come through the fires have changed.

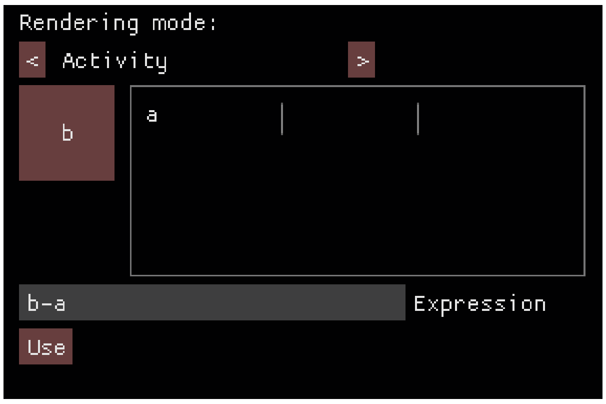


Figure 13 Displays the rendering mode menu that displays launch frequencies. Current values can be saved in variables (letters) by clicking the large red button. These values can then be processed through the arithmetic expression specified in the text field, the results of which are shown in the rendering of the synapses. It is also possible to restore the activity of the neurons (Ctrl and click the big red button), so a new average can be calculated.

2.5. Results

2.5.1. 3 neurons with 3 firemen

3 neurons are placed and connected according to Figure Figure 14. Each neuron has one fire er each, with the firing frequency 50 Hz. The frequency for the neurons firing scans with IDs 1 and 2, are offset by +2, and -2 ms respectively. All neurons are simulated at each run, and the rest of the settings have default values. The program is named "3_neurons_3_inputs.exe" in the folder that can be downloaded from the link in Appendix Appendix 2.

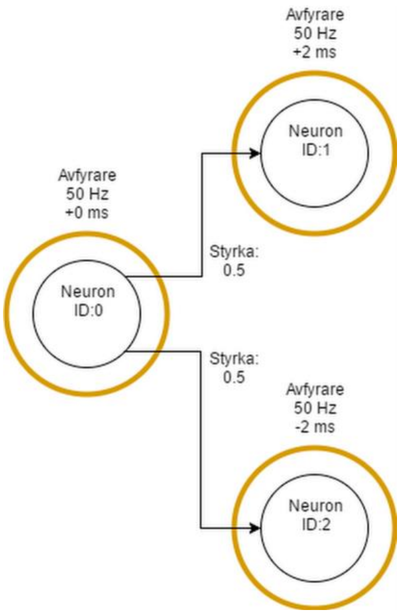


Figure 14 The neuron with ID 0 connects to the neurons 1 and 2 (one-way) with synapses of the strength 0.5. Each neuron has a 50 Hz frequency.

This configuration results in the synapse from neuron0 to 1 gradually increasing until the strength 1.0 is reached, and the synapse from neuron 0 to 2 gradually decreases until the strength 0.0 is reached. This takes an average of about 120 ms simulation time. Due to background activity, the neurons are sometimes activated randomly, which affects the synaptic forces, although in the end it does not change the final state.

2.5.2. 750 neurons with 1 fire

Here a fireman is placed on the coordinates (2.0, 0.0, 0.0), with radius 0.8 and a constant firing frequency of 35 Hz, in a brain with 750 neurons. All other settings have default values. The program is named "750_neurons_1_input.exe". The random variables are based on the current time, and will therefore vary from driving to driving. Therefore, this result part presents the normal behavior of the network, although the behavior of a specific run-time run may vary.

At the beginning of the simulation, the activity in the brain is only triggered by the background firings. After about 15 ms, however, an exponential reproduction of the impulses has been initiated, which in a short time activates most of the neurons in the network. During this stage, the majority of the positive synaptic forces are shifted to 0 (which can be read over time from the graph in Figure Figure 8 The activity of the neurons also decreases during a course of about 160 ms. At this time, the trigger's impulses propagate only a short distance before they die out. After 5 seconds of simulation time, the impulses spread from the beacon to the rest of the brain. There are also smaller areas of constant activity (see Figure Figure 15which then die out.

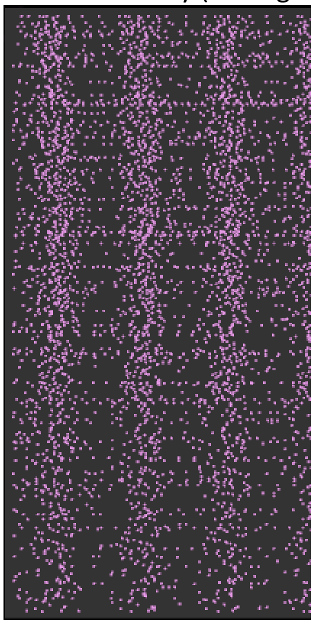


Figure 15 Raster graph after 5 seconds simulation time of network with a fire. The periodic pattern over time occurs when the launcher (35 Hz) sends impulses. The horizontal rows show neurons that are part of smaller self-supporting circuits, and whose activity is therefore constant.

2.5.3. 750 neurons with 3 firemen

This test has the same configuration as the one before, with different number and another behavior of the fire. The program is called "750_neurons_3_inputs.exe". Three fireers with radius 0.8 are placed in an equilateral triangle with a distance of 2 from the center of the network (origin). Fireers 0 and 1 always have the same firing frequency. The initial firing rates for launchrs 0 and 1, and 2, are a random value between 0 and 75 Hz. The learning factor (in$_g$ section 2.3Implementationbehaviour. During the period 10.0 to 10.2 seconds, all firing sequences will be 0, and between the period 10.2 and 10.6 seconds only the 2 frequency 50 Hz.

Here, too, after about 15 ms, the network turns to global constant activity, causing the synaptic forces to decline. The results of the measurements after 10 seconds are presented below.

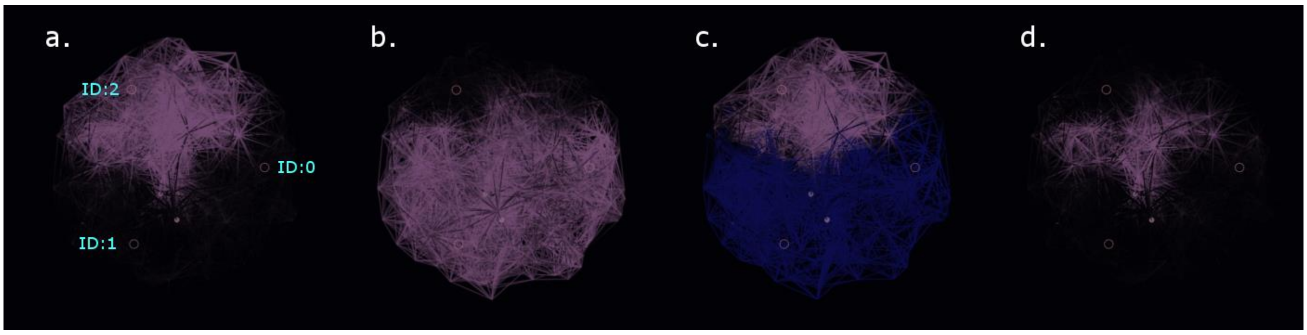


Figure 16 (a.) Shows brain activity in the period 10.2 to 10.6 seconds, when only the beacon with ID 2 is active (50 Hz). The ID of the firemen in relation to position also applies to the rest of the figure. (b)) Displays brain activity during the period 10.8 to 11.2 seconds, when the fires 0 and 1 are active (both 50 Hz). (c) Shows brain activity in a. subtracted with brain activity in b (the tool in Figure Figure 13 is used). Here, therefore, the neurons fired during only the period in a. are visible as pink, and those only during the period in b. as blue. (d.) Displays the task in a. multiplied by the task in b. This shows the areas that are active during both periods.

3. Analysis and discussion

The idea behind the simulation configuration in section 2.5.1 was to to some extent mimic the real experiment made in the report "Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type". Here, two physical neurons with a synapse between have been stimulated regularly and synchronized, with a certain interval time difference. A [8]positive time interval, when the presynaptic neuron is fired after the postsynaptic neuron, results in a gradual increase in the potential caused by the synapse in the postsynaptic neuron, and a negative time interval instead results in a decrease. In the simulation, the same thing is done, although for convenience it is the synaptic strength that is measured directly instead of the resulting potential. The results show that this simulation model is in line with reality (within the framework of this test). However, this does not mean that all aspects of plasticity are modelled. For example, in reality, the change in synaptic strength also depends on the synapse strength itself. A weak synapse has easier to become stronger than an already strong synapse. [8] I have chosen not to include this so as not to complicate the analysis and not to make it more difficult to balance the activity level of the brain. The behaviour of the inhibitory synapse is not based on any real study, but follows the same rules as the excitatory synapse. General and simple rules that always apply are in programming a good tactic to avoid bugs, and contribute to a more robust system that is often also easier to understand. However, this results in a dilemma when modeling biological systems. Evolution has an ability and a tendency to construct systems with a high complexity, which are thus difficult to model with accuracy. Therefore, i as a programmer need to make the assessment to what level of detail the simulation should model, as well as which parts of the biological systems can be simplified. Using a continuous system for synaptic forces, instead of simulating different types and quantities of neurotransmitters, is such a simplification. The digital domain often provides freer playing field than the physical, which means that digital systems often do not require the same complexity as the biological ones to achieve the same functionality. However, there are other limitations with the digital systems, such as computing power, which programmers need to take into account.

The simulation configuration in 2.5.2 shows that the network itself can counteract the feedback of impulses that initially causes the chaotic constant activity throughout the brain. It is the chaos that makes it impossible for a synapse to consistently fire before its target neuron, causing all synapses to weaken. It is also possible to see how the impulses from the launcher propagate to ever larger parts

[8]Guo-qiang Bi, Mu-ming Poo. Synaptic Modifications in Cultured Hippocampal Neurons:
Dependence on Spike Timing, Synaptic Strength, and
Postsynaptic Cell Type. The Journal of Neuroscience. 1998. PMID: 9852584.

of the network. Although there is always a synapse in each direction, STDP plasticity means that only one of these synapses will be strong, while the other will be weak. This means that the launcher's impulses are single-directed out of the launcher itself. The decision to add background activity came late in the work, but proved to be the key to getting the area that the fireer's impulses propagate to expand over time. Background firings in inactive or already active parts of the network rarely affect synaptic forces. However, if a background firing occurs in an inactive neuron at the edge of the firing range's active range, two possible things occur depending on whether the background firing occurs directly before or after the launcher's impulse arrives. If it occurs directly before, the synapse between the active and the inactive synapse weakens slightly, which in turn does not lead to anything. If it occurs immediately after, the synapse strengthens, which in turn can lead to the active neuron itself being able to fire the previously inactive neuron, which leads to further strengthening of the synapse. The net effect of this is that the background activity is more likely to result in the firing range becoming larger than smaller. However, only a few neurons, which are always active, create areas of only a few neurons. This is because the signals reconnect, and thus become self-supporting. Both the forming and destruction of these circuits seem to be triggered by the background activity, but how they work and can be counteracted needs to be further investigated. One ide is to introduce an energy system, which "exhausts" otherwise eternally active neurons.

The last experiment, in section 2.5.3aims to test the question: can the system build up an internal intuitive model of the relationships between its input values? The brain is given three inputs(through the firesna), two of which are linked(always fired at the same time). The fact that the two fires that are linked,, connected and isolated from the non-linking fireer (FigureFigure 16found and understands the connection. This confirms the question, although it doesn't say much about the extent to which the brain can actually do this.. Foretta example, in order to measure the number of inputs, it would be possible to increase the number of inputs. Would the brain sort out the connections between 10 firemen? It is also possible to include a type of relationship between theinput values, for example by delaying the time between two firetriggers being activated, or by utilizing then temporal encoding that thesystem allows. In principle, this network should be able to find a wide range of different types of causation. By entering pixel values from an image as inputs, the network would likely learn to classify between different objects in the image. One obstacle to this, however, is becount intensity,which increases sharply with the number of neurons. A variety of computational optimizations could performs,although sometimes it would be at the expense of the biologicalmodeling's true to life. Implementing a kind of reward system would be a given next step for this project,as this would provide the ability to let the brain control virtual agents to solve problems other than finding connections.. Another possible direction is to train a kind of general formation in a given brain, which was the original intention of the work.. This system would then become a way to allow computers themselves to develop a basic intuitive understanding of concepts such as physics and psychology.

4. Source list

Guo-qiang Bi, Mu-ming Poo. Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and
Postsynaptic Cell Type. The Journal of Neuroscience. 1998. PMID: 9852584.

Hebb O. Donald. The Organization of Behaviour. New York: Wiley & Sons. 1949.

Hebb's Three Postulates: from Brain to Soma. [online video]. 2015.
https://www.youtube.com/watch?v=SIp_CTEfiR4 (Hämtat 2016-10-30)

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, Samuel J. Gershman. Building Machines That Learn and Think Like People. CBMM. 2016. arXiv:1604.00289v2

H. Markram, W. Gerstner, P. J. Sjöström. Spike-Timing-Dependent Plasticity: A Comprehensive Overview. Front Synaptic Neurosci. 2012. doi: 10.3389/fnsyn.2012.00002
[1] CCNLab, CCNBook/Networks, https://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Networks, 31 Mars 2015, (Hämtad 2016-11-20)

Khanacademy. Neuron action potentials: The creation of a brain signal, https://www.khanacademy.org/test-prep/mcat/organ-systems/neuron-membrane-potentials/a/neuron-action-potentials-the-creation-of-a-brain-signal (Hämtat 2017-02-28)

Adam Spector. 2015. Spotify Just Dove Deep Into Machine Learning Personalization. LiftIgniter. 28 Maj. http://www.liftigniter.com/spotify-just-dove-deep-into-machine-learning-personalization/ (Hämtat 2016-10-30).

5. Attachments

Appendix 1 - GitHub Project Link

https://github.com/Axelwickm/NeuroCorrelation/tree/Snapshot

Appendix 2 - Pre-packaged example download

https://drive.google.com/drive/folders/0B56HcM6Y9mY5VkNWQnY4S0xCRms?usp=sharing

Appendix 3 - A simple program that first creates a brain with 100 neurons, and then opens a window indicating when this brain is rendered.

```
#include <NeuCor.h>
#include <NeuCor_Renderer.h>

int main(){

    // Initerar hjärnobjekt med 100 neuroner
    NeuCor brain(100);

    // Skapar en array med 3 float-värden
    float inputs[] = {20.0, 2.5, 150.0};

    // Ger denna array till hjänan
    brain.setInputRateArray(inputs, 3);

    // Sätter tidsteg till 0.05 ms per körning
    brain.runSpeed = 0.05;

    // Gör så att alla neuroner alltid simuleras vid körning
    brain.runAll = true;

    // Initierar renderare och ger den minnesadressen till hjärnan
    NeuCor_Renderer renderer(&brain);



    // Stiger in i en loop som körs så länge som fönster ska vara öppet
    while (true){

        // Simulerar hjärnan
        brain.run();

        // Hämtar användarhandlingar
        renderer.pollWindow();

        // Ritar ut på skärmen
        renderer.updateView();
    }

    return 0;
}
```