Polhemsskolan

Neurokorrelation

Simulation av neuronnätverk

Axel Wickman
Gymnasiearbete 100 poäng
Klass TeInf3a
Teknikprogrammet
Läsåret 2016/2017
Handledare: Mikael Bondestam

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.

Innehåll

1. INLEDNING **1**
1.1. Syfte och Frågeställning 2
1.2. Materiel och metod 2
1.2.1. Nätverksmodell *2*
1.2.2. Programmering *3*
2. UNDERSÖKNING OCH RESULTAT **4**
2.1. Bygga programmeringsmiljö 4
2.2. Struktur och grundläggande system 4
2.3. Implementering 6
2.4. Användning 11
2.4.1. Bibliotek *11*
2.4.2. Grafiskt gränssnitt *13*
2.5. Resultat 16
2.5.1. 3 neuroner med 3 avfyrare *16*
2.5.2. 750 neuroner med 1 avfyrare *16*
2.5.3. 750 neuroner med 3 avfyrare *17*
3. ANALYS OCH DISKUSSION **18**
4. KÄLLFÖRTECKNING **19**
5. BILAGOR **21**
Bilaga 1 - GitHub projektlänk 21
Bilaga 2 - Förkomplilerade exempel nerladdning 21
Bilaga 3 - Ett enkelt program som först skapar en hjärna med 100 neuroner, och sedan öppnar ett fönster som visar när denna hjärna renderas. 21

1. Inledning

De senaste åren har flera stora framsteg gjorts inom det vetenskapliga området maskininlärning. Ökad datorkraft, mer investeringspengar från företag, samt ett större allmänt intresse gällande området, har bidragit till detta. Det finns en stor entusiasm bakom möjligheten att få datorer att tänka mer som människor, då det gör det möjligt att eliminera mänskliga tillkortakommanden ur vissa arbetsuppgifter. En tänkande dator kan jobba dygnet runt, och är ofta billigare, säkrare, och snabbare än en tänkande människa. Redan idag så ser vi maskininlärningsalgoritmer i full användning. På internet används de för att leverera passande sökresultat och annonser, samt rekommendera filmer, produkter och låtar till användaren[1]. Detta görs genom att algoritmen tar fram resultat med hänsyn till viss information, och sedan förändras beroende på hur väl dessa resultat presterar. Prestationen kan mätas på olika sätt beroende på tillämpningsområde. Om man har en musiktjänst som rekommenderar låtar, så kan algoritmen ta hänsyn till exempelvis användarens lyssningsmönster, ålder, tid på dagen och nuvarande väder. Dessa variabler ger sedan ett antal låtförslag, via algoritmen. Prestationen kan mätas genom, exempelvis, hur länge användaren valde att lyssna på de rekommenderade låtarna. I detta fall skulle algoritmen utvecklas mot att rekommendera låtar som användare lyssnar länge på.

Några andra användningsområden för maskininlärning är diagnostisering av sjukdomar, datorseende, taligenkänning, språkförståelse, robotlokomotion, kontrollsystem och Artificiell intelligens (AI) ibland annat spel. Algoritmerna bakom dessa områden varierar dock drastiskt. Artificiella neuronnätverk (ANN) används i många fall på grund av deras förmåga att hitta komplexa samband. De består av neuroner som kan kommunicera med varandra med elektriska signaler via kopplingar. Varje koppling går från en neuron (den presynaptiska cellen) till en annan (den postsynaptiska cellen), och har en viss styrka som påverkar amplituden av de signalimpulser som passerar igenom. När en neuron får signaler via kopplingar så bearbetas de och förmedlas vidare. Inlärning sker då sedan genom att variera till exempel styrkan på kopplingarna, eller variera hur varje neuron behandlar signalerna. Ju fler neuroner och kopplingar ett nätverk har vars värden inte bestäms av en människa, ju ”djupare” är det. Det finns många typer av neurala nätverk. Bland annat kan strukturen på nätverken kan variera. I den vanligaste typen är neuronerna organiserade i lager, och signalerna kan bara röra sig framåt. Detta är ett feed-forward nätverk, i kontrast till återkopplande nätverk. Man kan ha olika typer av signaler i nätverken. Ofta är det bara ett enkelt nummervärde, men det finns undantag som i faltnings-neuronnätverk (CNN), som har visat sig vara väldigt bra på bland annat datorseende. Dessa nätverk är inspirerade av syncentrum i våra egna hjärnor (även om de flesta studier gjordes på katter), och har varit en stor utvecklingskälla inom maskininlärningen de senaste åren.

En inspirationskälla till mitt projekt var publikationen ”Building Machines That Learn and Think Like People”[2]. Här diskuteras problemet om varför vi människor kan lära oss att utföra uppgifter som att spela ett spel på bara ett antal försök, samtidigt som det kan ta hundratals timmar för ett artificiellt neuronnätverk att komma i närheten av vår prestationsförmåga. Författarnas uppfattning är att det beror på grundläggande skillnaderna mellan de artificiella nätverk som används och utvecklas idag, och hur vi människor fungerar. De föreslår att vi människor utnyttjar den kunskap om världen vi redan har, och applicerar den i det nya sammanhanget. Detta låter oss göra antaganden som det artificiella nätverket måste testa sig fram till. Vanligtvis när man vill lära upp ett nätverk så börjar man nämligen med att det är helt otränat, utan någon erfarenhet alls. Genom att först låta ett neuronnätverk bygga upp intuitiva modeller om till exempel fysik och psykologi, så ger man det en grund att basera sin utveckling på när det får ett nytt problem att lösa, menar författarna. Detta kallas modell-baserad

[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/ (Hämtat 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

inlärning, för att nätverket bygger fram en lösning baserat på sin modell av problemmiljöns uppförande.

Att bygga ett sorts nätverkssystem som utnyttjar dessa teorier om modellbaserad inlärning, och sedan träna upp det, tror jag skulle vara ett stort steg mot allmänt AI - där varje tränat nätverk (hjärna) används till mer än att bara lösa enstaka problem. Ett första steg skulle vara att skapa ett nätverk som automatiskt bygger upp en intuitiv modell av de värden man ger det. Med intuitivt menas att det inte finns någon resonemangsförmåga i nätverket gällande de samband det har lärt sig, vilket är en följd av att bara grundläggande inlärningsmekanismer kommer användas. Att få nätverket att styra mot att lösa problem efter hur väl jag som programmerare värderar dess prestation ingår ej i detta projekt, men är ett möjligt framtida forskningsområde.

1.1. Syfte och Frågeställning

Syftet med detta projekt är att ta reda på hur man kan bygga upp en datorsimulation av en biologisk hjärna, som sedan kan användas för maskininlärning. Frågeställningen är hur man kan gå till väga för att bygga detta system, samt om en biologisk modell fungerar för att bygga upp en intern intuitiv modell av sambanden mellan inputvärden.

1.2. Materiel och metod

1.2.1. Nätverksmodell

Den mänskliga hjärnan är ett bra exempel på den sortens system som jag vill bygga. Man kan se den som en korrelationsmaskin. Den tar inputs i form av nervsignaler från våra sinnen, och hittar samband mellan dessa. Dessa samband tas fram genom relativt enkla regler som fungerar på cellnivå, och innebär att kopplingarna mellan neuroner, där det finns en korrelation i avfyrningsmönstren, blir starkare. Reglerna teoretiserades av den kanadensiska forskaren Donald Hebb 1949 i sin, inom neuropsykologin, mycket kända bok ”*The Organization of Behaviour*”[3]. ”Hebbianska” inlärningsmodeller är mycket vanliga inom artificiella neuronnätverk, även om deras implementering av nödvändighet varierar beroende på nätverkstyp. Den Hebbianska inlärningsmodellen påverkar bara styrkan av kopplingarna mellan neuronerna, och kan sammanfattas: Celler som avfyras tillsammans, sammankopplas. Om en nervcell vid upprepande tillfällen orsakar att en annan avfyras, så stärks kopplingen mellan dem. Detta kallas Long-term Potentiation (LTP).[4] Motsatsen till detta kallas Long-term Depression (LTD). Denna förmåga att förändra styrkan i kopplingarna kallas synaptisk plasticitet.

Det finns två populära sätt att representera biologiska neuroner i nätverksmodeller. Man kan till exempel representera nervcellers avfyrningsfrekvens, en s.k. rate-modell. Denna modell är robust för inlärning, lätt att implementera, och är ofta biologisk representativ, men den klarar inte av att representera alla sorters informationsöverföring. Neural kodning är hur information kan representeras och förflyttas i hjärnan. En rate-modell låter information bara representeras i hur aktiva neuronerna är. Eftersom syftet med detta projekt är att skapa en relativt verklighetstrogen biologisk modell, så är det nödvändigt att tillåta temporal-kodning. Detta genom att istället representeras den elektriska spänningspotentialen, i både neuronerna och kopplingarna (som hädanefter hänvisas till som synapser). Över tid kan spänningen förändras. Om spänningen överstiger en viss tröskelpotential, så orsakar det en nervimpuls, som förmedlas vidare till synapser och därmed andra neuroner. Detta gör modellen till ett impuls-baserat nätverk (SNN). Denna temporal-kodning innebär att en enstaka neuron kan överföra information via specifika avfyrningssekvenser, vilket i praktiken innebär att den får den tekniska möjligheten att kommunicera med till exempel morsekod. Att använda ett SNN möjliggör att även använda en mer realistisk modell för den synaptiska plasticiteten. Impuls-tidsbaserad-plasticitet

---

[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 (Hämtat 2016-10-30)

(STDP) baserar förändringen i synapsens styrka på den relativa tiden mellan nervimpulser. Denna modell är i full enlighet med Hebbs regler, och har visat sig representera den biologiska verkligheten ganska väl.[5]

Som tidigare nämnt så kan neuronnätverk bestå av olika sorters strukturer. I detta projekt placeras neuronerna ut spatialt i 3D utrymme, och skapar kopplingar till alla de andra närmaste neuronerna. Koordinaterna är slumpmässigt genererade decimaltal, som är jämt fördelade inom en sfär. Detta system innebär att hjärnan har en återkopplande neuronstruktur. Här uppstår problemet med positiva återkopplingsloopar, då signaler i hjärnan kan livnära sig själva obegränsat länge. Detta kan ske i även verkligheten under vissa genetiska förutsättningar, och är känt som epilepsi. Ett av hjärnans sätt att motverka detta är genom att placera ut inhibitoriska synapser, som hämmar aktivitet i neuroner. Denna sorts synapser är oftast mer lokaliserade, har kortare kopplingar än excitatoriska synapser (som främjar aktivitet i neuroner). De är även ovanligare, då endast 15 % av neuronerna i hjärnbalken har en inhibitorisk roll. Deras roll i faktisk inlärning och informationsbearbetning är omstridd inom forskarvärlden, men det står klart att de har en viktig funktion inom att motverka epileptiska anfall.[6]

I och med att hjärnan inte har någon mekanism som ”motiverar” den att sträva efter vissa mål, så är det poänglöst att bygga ett test som mäter hjärnans prestation. Skulle detta vara möjligt så hade man kunnat träna hjärnan, och sedan belönat förutsägningsförmågan, vilket man skulle extrahera som nervsignaler i vissa områden. Denna lösning hade dock haft nackdelen att man inte skulle veta hur mycket av den förutsägningsförmåga som uppstår som på grund av synapsernas naturliga (STDP) tendens att hitta samband, och hur mycket som bara är en produkt av belöningssystemet. Av denna anledning väljer jag därför att mäta hjärnans inlärning genom direkt analys av nätverket. En renderingsmotor är här mycket hjälpsamt då det blir möjligt att snabbt och enkelt göra en intuitiv bedömning av hjärnans beteende och förmåga, vilket är fördelaktigt under utvecklingen och designen av simulationssystemet. Genom andra verktyg går det sedan att göra en mer objektiv och vetenskaplig bedömning av systemet. Till exempel går det att observera vilka områden är aktiva när specifika inputs ges.

1.2.2. Programmering

Programmeringsspråket som väljs sätter grunderna för strukturen på programmet, hur utförlig koden behöver vara, projektets arbetsflöde, var programmet kan köras, samt hur snabbt koden kör. Eftersom att koden i detta sammanhang skulle användas i en simulation som med tid utvecklas mot ett visst tillstånd, som sedan analyseras, så är det fördelaktigt att detta tillstånd nås så snabbt som möjligt i förhållande till realtid. Därför krävdes ett snabbt språk, vilket oftast medför ett lågnivåspråk, där programmeraren har stor makt och ansvar att definiera algoritmerna i detalj genom att jobba nära hårdvaran. Bäst lämpat för detta är C++, som även har fördelen att det fungerar väl i 3D miljöer. C++ är en vidareutveckling på C språket, och har fler inbyggda funktioner samtidigt som det ger mycket kontroll. Språket har stöd för objektorienterad programmering, vilket gör det passande i större simulationsprojekt, där man har många olika sorters objekt som alla har delat beteende.

Eftersom att C++ är ett språk som behövs kompileras före att det körs, så krävs att man väljer en kompilator. Kompilatorns roll är att översätta programmerarens kod i C++ till den maskinkod som körs på processorn. I detta projekt används MingW-kompilatorn på Windows. Detta för att kompilatorn är väl integrerad med programmeringsmiljön CodeBlocks. Vid installation av CodeBlocks kan väljas att även installera MingW. För att hantera 3D grafiken i renderaren så krävs ett API (applikationsprogrammeringsgränssnitt) som dels kan tala med hårdvaran, och dels kan fungera på ett brett spektrum av mjukvarumiljöer. Här väljer jag OpenGL, vilket är en sorts standard som hårdvarutillverkare har implementerat i sina enheter. Denna standard gör det möjligt för samma kod att

---

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

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

köra och fungera (approximativt) lika på olika system. OpenGL ansvar dock endast för att rendera pixlar på en skärmyta, det krävs ett annat API för att prata med operativsystemet för att kunna skapa ett fönster. Här så använder jag GLFW, som är plattformsoberoende (fungerar på flera operativsystem), och kan hantera och vidarebefordra användarinmatning till resten av programmet.

2. Undersökning och resultat

2.1. Bygga programmeringsmiljö

För att hantera alla filer under projektets gång så användes GitHub (projektlänk finns under bilagor). Detta gör det möjlig att ha kontroll över förändringshistoriken, dela min kod, och ha uppdaterade säkerhetskopior på en server. Det krävdes även ett antal externa referenser, och det krävdes därför extra arbete för att konfigurera programmeringsmiljön. Efter att CodeBlocks och MingW hade installerats så laddade jag hem och installerade dessa externa referenser, vilka är s.k. bibliotek som innehåller kod som andra projekt kan importera och använda.

Tabell 1 Visar de namnet av bibliotek som används i koden, samt vilken funktion de spelar.

Bibliotek	**Användning**
GLFW (& OpenGL)	Skapa fönster och rendera 3D grafik.
GLM	Förvara och bearbeta matriser och vektorer
GLEW	Hantering av OpenGL tillägg
PicoPNG	För att ladda in bilder från .png format
dear imgui	Programmering av användargränssnitt
Boost	Stable vector -behållare
TinyExpr	Tolka och beräkna aritmetiska formler

MingW-mappen, vars plats valdes under installationen av CodeBlocks (standarplats är i CodeBlocks-mappen) har en undermapp med namn "bin". Denna mapp bör läggas till i miljövariabeln PATH, då den innehåller flera program som behöver anropas under installation av biblioteken. Installationen sker genom att flytta mappen med källfilerna (.h, .cpp, eller .hpp) till under "includes" i MingW mappen. Installationen av dessa bibliotek varierar och sker på följande vis:

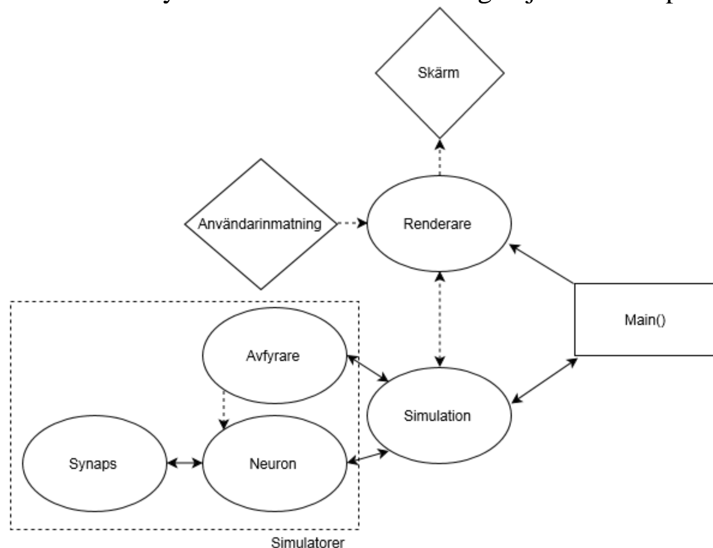
1. PicoPNG, dear imgui, TinyExpr - Redan inbäddade i projektfilerna och följer ner när Git-projektet laddas ner.
2. GLM, Boost - "glm", samt "boost" -mapparna läggs som undermappar i "includes"-mappen i MingW-mappen. Detta tillåter kompilatorn att hitta och använda filerna när programmet skapas.
3. GLFW – Laddas ner förkompilerat för Windows 32 bitars. I den nerladdade ZIP-filen finns en mapp med namn ”include” som slås ihop med den i MingW. De förkompilerade filerna finns under ”lib-mingw” i ZIP-filen. Här ska de två .a-filerna läggas till under ”lib” i MingW. Den kvarvarande .DLL-filen ska läggas på ett ställe känt för projektets .exe-fil efter kompilering, till exempel i samma mapp som den, eller i MingWs ”bin”-mapp som lades till i PATH-variabeln tidigare.
4. GLEW – Finns i förkomplilerade versioner, men dessa fungerar ej för MingW-32, vilket betyder att kompileringen behövs göras manuellt. Först bör källfilerna laddas ner (finns på hemsidan), och extraheras till sin egen mapp. Sedan kan kommandotolken öppnas i mappen (Ctrl + högerklick), och följande kommandon köras:
    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

Detta skapar tre filer i GLEWs ”lib”-mapp med två .a- och en .DLL-fil som kan installeras på samma sätt som de i GLFW. Det finns även en ”includes”-mapp som bör slås ihop med den i MingW.

2.2. Struktur och grundläggande system

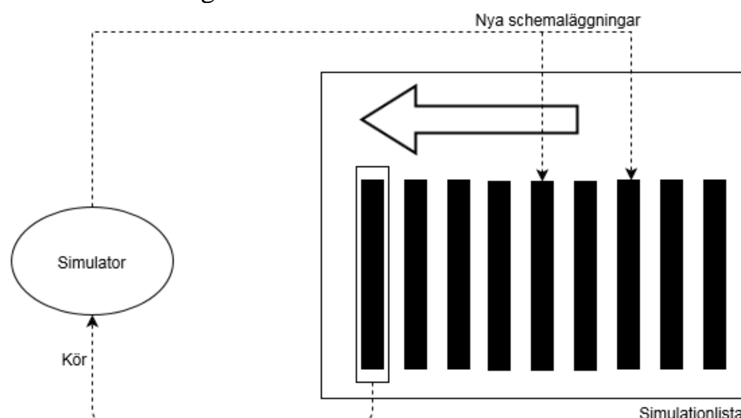
Strukturen på programmet är utformad på sådant sätt att simulationsobjektet (själva hjärnan) kan vara självständigt från renderaren, som bara ges minnesadressen till simulationen. Renderaren kan på så sätt extrahera information som sedan bearbetas om till pixelvärden som sedan skickas till skärmen.

Renderaren kan via denna minnesadress även ändra på simulationen. Detta ger användaren möjlighet att förändra till exempel simulationshastighet under körtid. Simulationsobjektet förvarar olika typer av simulatorer. Gemensamt för dessa simulatorer är att de alla känner till simulationsobjektet, och att de har en körfunktion som uppdaterar objektet i förhållande till simulationens tidsvariabel. Avfyrar-simulatorn ser till att avfyra vissa neuroner efter en viss frekvens som definieras i main-funktionen. Det är detta system som används för att ge hjärnan de input-värden som sedan bearbetas.



Figur 1 Visar de funktionella förhållandena mellan de olika klasserna (ovaler), användargränssnitt (romber), och main-funktionen (rektangel). Streckad linje visar att informationsutbyte sker. Heldragna pilar mellan klasser visar att klassen till höger äger klassen till vänster. Klasserna inom den streckade rektangeln ärver alla av simulator-klassen.

Simulationsobjektet äger en simulationslista vars uppgift det är att köra alla simulatorer vid rätt tillfälle i rätt ordning. Varje simulator ansvarar för att schemalägga tillfällen att köras i framtiden i denna globala lista. Varje tillfälle sorteras automatiskt så att det som ligger närmast i tid ligger först. Efter att ett tillfälle har passerat och simulatorn har körts så raderas elementet ur listan. Detta system innebär i praktiken att bara det som är nödvändigt att simulera behöver simuleras, och så att simulationshastigheten kan variera utan att simulationens beteende förändras.



Figur 2 Visar hur simulationslistan anropar körfunktionen av simulatorn som det tillfälle som ligger närmast i tid (längst till vänster) länkar till. Simulatorn kan då komma att schemalägga senare simulationstillfällen i framtiden, som då sorteras in beroende på tid i simulationslistan. Efter att ett simulationstillfälle har fått körfunktionen anropat så raderas det, och nästkommande tillfälle blir då först i listan.

Genom att utveckla både simulationen och renderaren parallellt så blir det möjligt att enklare reda ut buggar, och förstå hur systemet beter sig under olika förhållanden. I C++ är det vanligt att klasser och globala variabler först deklareras i s.k. header-filer (.h), för att sedan definieras i själva kod-filerna (.cpp). Detta bidrar till bättre översikt av projektet, och gör det enklare att använda klasserna i andra sammanhang utan att komma i kontakt med den interna koden som klassen själv använder. I detta projekt finns det främst två viktiga header-filer. ”NeuCor.h” deklarerar följande klasser:

```
// 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 {
```

Kodavsnitt 1 Visar vilka klasser och strukturer NeuCor.h deklarerar.

”NeuCor_Renderer.h” deklarerar:

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

Kodavsnitt 2 Visar vilka klasser och strukturer NeuCor_Renderer.h deklarerar.

Genom att dela upp simulationsklasserna och renderaren för sig så blir det möjlig att bara inkludera simulationen i sitt projekt, om det bara är den som är nödvändig. Detta innebär att man i så fall inte behöver installera alla bibliotek som renderaren behöver.

2.3. Implementering

När man i C++ skapar ett objekt av en viss klass så anropas en konstruktor, en sort initieringsfunktion vars uppgift det är att konstruera objektet. De i header-filen definierade variablerna finns redan i det nya objektet, men dom har inte nödvändigtvis några tilldelade värden. I mitt projekt innebär det att hjärnan är tom när den först skapas, och att man i konstuktorn (eller senare) behöver placera ut neuroner och synapser. Detta gör jag genom att ge hjärnans (NeuCor-klassens) konstruktor ett heltal som in-parameter, vilket anger hur många neuroner (med delvis slumpmässiga egenskaper) som initialt ska placeras ut. Detta görs i två steg: först skapas och förvaras alla neuronerna, sedan skapas kopplingarna (synapserna). Kopplingarna skapas genom att varje neuron mäter avståndet till de andra neuronerna, och om detta avstånd är mindre än 1 längdenhet, samt kopplingen inte redan finns, så skapas en synaps som går från den presynaptiska neuronen till den postsynaptiska neuronen. Denna synaps förvaras i en behållare inuti presynaptiska neuronen. Denna metod innebär att det alltid kommer finnas två kopplingar åt vardera riktning när två neuroner är tillräckligt nära varandra. Algoritmen är dock ineffektiv då den måste jämföra avstånden till varje neuron för varje neuron, vilket skapar en exponentiell tillväxt i antalet beräkningar som måste genomföras i och med att neuronantalet ökar (en s.k. $O(n^2)$ – algoritm inom Big O notation). Detta problem kan i framtiden lösas genom att placera koordinaterna av varje neuron i ett kd-träd, en binärt partitionerande behållare som minskar antalet nödvändiga jämförelser linjärt med antalet neuroner. Detta ligger dock utanför ramarna av detta projekt. När neuron- och synapsobjekten skapas så anropas även deras konstruktorer, som i båda fall ansvarar för att definiera objektens egenskaper, så som: vilopotential, kopplingsstyrka, form på impulsspänning över tid, etc. Synaps-konsturkorn lägger även en identifierare av sig själv i en särskild behållare i den postsynaptiska neuronen, vilket gör så att denna neuron vet vilka input synapser den har, även om den inte äger dem.

Neuroner kan skapas indirekt i hjärnan utanför konstuktorn genom att anropa en av hjärnobjektets funktioner:

```
// 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();
```

Kodavsnitt 3 Visar hur det går att skapa neuroner i main-funktionen genom hjärnobjektet.

När NAN (not-a-number) anges så skapas koordinaterna istället slumpmässigt, jämt fördelade inom en sfär. Detta görs genom att 3 slumpmässiga reella tal (x, y, z) genereras inom ett visst omfång. Om dessa tal, när de tolkas som 3D-koordinater, befinner längre än ett visst avstånd från origo, så förkastas talen, och tre nya genereras, annars så används talen som neuronens koordinater. Denna algoritm är enkel att implementera, men relativt ineffektiv då den förkastar en del av koordinaterna. Sannolikheten att koordinaterna inte behöver förkastas kan beräknas genom:

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

Där r är det maximalt tillåtna avståndet från origo.

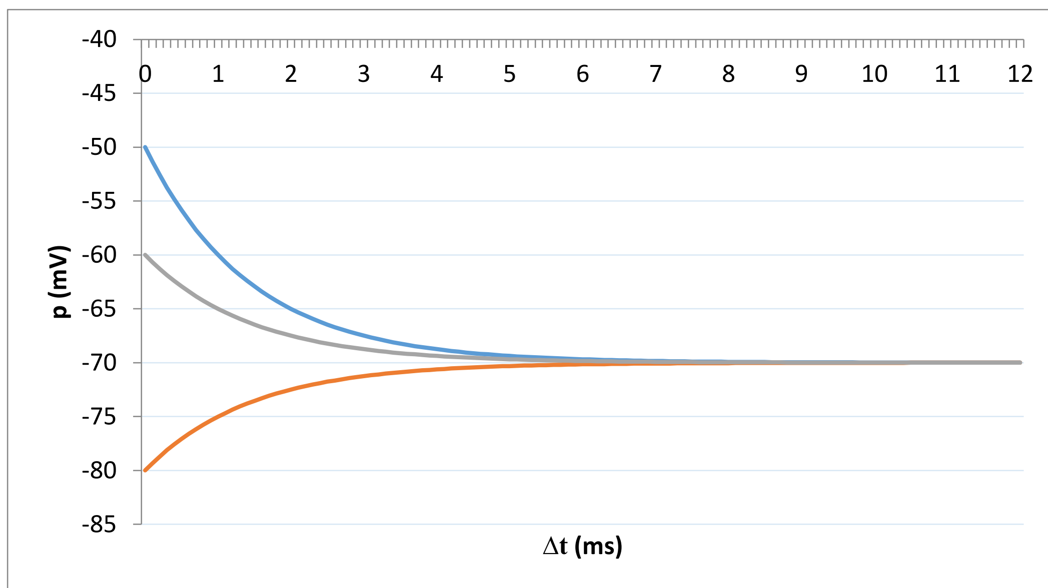
Efter detta är hjärnan initierad, och redo att simuleras. Det är här nödvändigt att skiva koden till de tre simulatorerna, samt hjärnobjektet. I hjärnans globala körfunktion implementeras bakgrundsaktivitet genom att varje neuron får en $1/600$ per ms chans att avfyras spontant. Neuronens körfunktion anropas regelbundet av bland annat hjärnobjektet (genom simulationslistan). Denna funktion uppdaterar neuronens tillstånd beroende på hur mycket tid som har passerat sedan den senast anropades.

```cpp
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));
}
```

Kodavsnitt 4 Visar utseendet av neuron-klassens kör-funktion.

Aktivitets-variabeln, som håller reda på avfyrningsfrekvens, beräknas även. De fyra andra funktionerna som anropas reglerar neuronens nuvarande elektriska potential. Medlemsfunktionen (d.v.s. en funktion som endast tillhör klassen) ”charge_insynapses” är neuronens förbindelse med sina in-synapser. Denna funktion ökar neuronpotentialen med: $\sum_{i=0}^{a}(f_i * \Delta t * g(\Delta t))$ där a är antalet aktiva synapser, Δt är tid sedan senast uppdatering, f_i är impulsstyrkan från en specifik synaps, och $g(\Delta t) = 0.9943 * e^{0.3702 * \Delta t}$ är en funktion framtagen genom regression som gör så att beteendet är desamma oavsett Δt. En synaps räkans som aktiv i 2 ms från början av en impuls. Denna funktion är en kraftigt förenklad modell av hur dendriter tar upp neurotransmittorer i det synaptiska gapet, och hur dessa signaler sedan summeras i neuronen. Istället för att integrera en impulsspänningens förändring över tid så integreras endast ett konstant värde över tid. Detta förenklar de matematiska beräkningarna som datorn behöver utföra och gör hjärnans beteende mindre invecklat.

Efter att in-synapsernas impulser har integrerats och summerats så anropas ”charge_passive”, vars uppgift det är att få neuronens potential att gå mot sin vilopotential (b), vilket är -70 mV, över tid. Detta gör på exponentiellt vis genom funktionen $p_{uppdaterad} = (p_{nuvarande} - b) * c^{\Delta t} + b$, där c är förändring per ms, vilket i simulationen är 0,5.



Figur 3 Exemplet visar att potentialen (p) alltid går mot vilopotential över tid sedan senast körning (Δt) oavsett startvärde.

Nästa funktion, ”charge_thresholdCheck” kontrollerar om huruvida neuronens potential är över en viss tröskelpotential (-55mV). Om det är fallet och neuronen eller synapsen inte redan håller på att avfyras, så anropas neuronens avfyrningsfunktion. Denna funktion utför ett antal uppgifter. Den ökar neuronens avfyrningsräknare (som används för att avgöra aktiviteten) med 1, sätter spår-variabeln till 1, och sätter neuronens variabel för senaste avfyrning till den nuvarande tiden. Utöver detta så avfyrar den STDP-inlärningsalgoritmen i alla in synapser, samt anropar alla ut-synapsers avfyrningsfunktion med impulsstyrkan som parameter.

Den sista funktionen som ändrar potentialen i neuron-klassen är ”AP”. Denna funktion ger formen på neuronens spänning under en nervimpuls. För att förstå anledningen till denna form, så är det nödvändigt att förstå den underliggande biokemiska processen bakom. Med begreppet potential menas i detta sammanhang skillnaden på den elektriska spänningen inuti och utanför en neurons cellmembran. Spänning uppstår på grund av olika jonkoncentrationer, vilket är ett resultat av de natrium-kaliumpumpar finns i cellmembranet. Dessa proteinpumpar jobbar för att bibehålla natriumjoner (Na^+) utanför cellen och kaliumjoner (K^+) inuti cellen. Utan dessa skulle jonkoncentrationerna utanför och inuti cellen bli desamma på grund av att membranet är super permeabelt för kaliumjoner och delvis permeabelt för natriumjoner. Det är detta som får membranpotentialen att gå mot vilopotential (-70 mV). En nervimpuls är en hastig och skarp förändring i jonkoncentration, och därmed också den elektriska potentialen. Denna förändring uppstår när spänningsstyrda jonkanaler, som även de finns i membranet, öppnar och stänger flödet av joner beroende på den aktuella spänningen i membranet. När excitatoriska in-synapser aktiveras, så flödar det in positivt laddade joner i neuronen, vilket ökar den elektriska spänningen. Om den elektriska spänningen överstiger en viss tröskelpotential (-55 mV) så öppnas de spänningsstyrda natriumkanalerna som normalt sätt är stängda, vilket gör så att det flödar in natriumjoner in i cellen. Detta får membranpotentialen att stiga (till ca +30 mV) och kallas depolarisering. Vid detta tillfälle börjar natriumkanalerna att stänga, och natriumjonerna börjar pumpas ut ur cellen igen på grund av natriumpumparna. Samtidigt så börjar de annars stängda spänningsstyrda kaliumkanalerna öppnas, vilket orsakar att kaliumkoncentrationen inuti cellen minskar, och därmed också cellens membranpotential. Detta kallas repolarisering. När vilopotential nås, så stängs inte dessa kaliumpumpar omedelbart, utan det fortsätter läcka kaliumjoner ur cellen. Detta gör att membranpotentialen temporärt understiger vilopotentialen, vilket kallas hyperpolarisering.

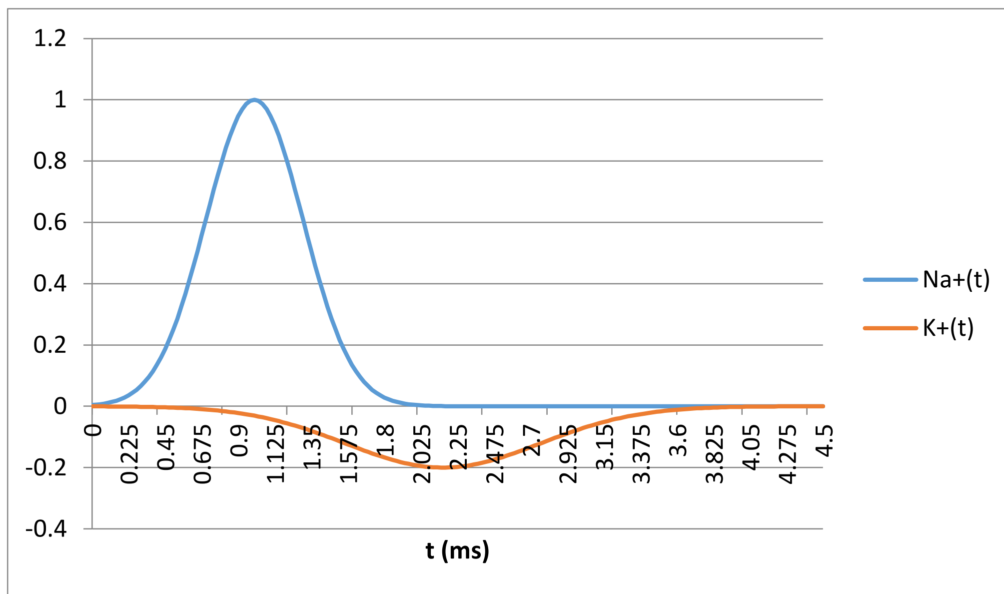
Kaliumpumparna återställer sedan kaliumjonskoncentrationen, och neuronen har då helt återställda jonkoncentrationer samt en återställt membranpotential.[7]

Detta biologiska uppförande modelleras i AP-funktionerna, men kommer här representeras med matematisk notering. Värdet som funktionen V(t) ger (där t är tiden i ms sedan senast avfyrning påbörjades), blir neuronens potential om en avfyrning är pågående. Genom att simulera den relativa mängden natrium- och kaliumjoner över tid, och sedan summera dessa två mängder, så går det att avgöra hur neuronens potential förändras över tid. Min bedömning har här varit att använda Gaussfunktion för att modellera de relativa jonmängderna under en nervimpuls.

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

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

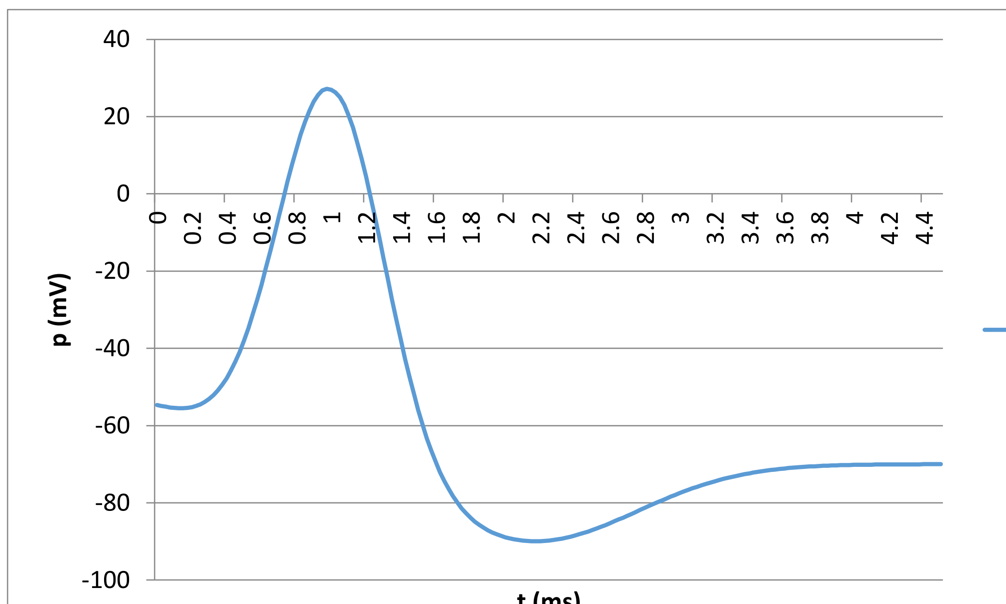
Även här är t tid sedan senast avfyrning påbörjades. Jag har valt att i simulationen använda värdena:
$a_1 = 1;\ d_1 = 1;\ w_1 = 0.3;\ a_2 = -0.2;\ d_2 = 2.16;\ w_2 = 0.6$
Observera att värdet a_2 är negativt, då den relativa mängden K^{+} innanför cellmembranet först minskar, och sedan ökar.



Figur 4 Visar hur den relativa mängden av Na⁺- och K⁺-joner förändras över tid sedan början av impuls (t) i förhållande till varandra.

Genom att utgå från dessa två funktioner går det att konstruera V(t):
$V(t) = f * \left(Na^{+}(t) + K^{+}(t)\right) + b + (T - b) * \max\,(1 - t;\ 0)$
Här är b vilopotential (-70 mV), T tröskelpotentialen (-55 mV), och f är 100. Den sista termen $(T - b) * \max\,(1 - t;\ 0)$ adderas för att åstadkomma att impulspotentialen börjar på tröskelpotentialen istället för vilopotentialen. Funktionen max returnerar det högsta värdet av de invariabler som anges. Uppmärksamma att funktionen V(t) bara gäller i 2 ms från början på avfyrning. Detta på grund av den bedömning jag har gjort att en impuls räknas som pågående i 2 ms (med dessa valda konstanta värden). Efter denna tid är det möjligt för neuronen att åter igen avfyras.

[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 (Hämtat 2017-02-28)



Figur 5 Visar hur potentialen (p) förändras över tid sedan början av impuls (t) när funktionen V(t) används.

När en neuron avfyrar en synaps, vilket sker i början av en impuls, så räknar synapsen i förtid ut hur lång tid det kommer att ta för impulsen att nå fram till målneuronen genom att multiplicera synapsens signalutbredningshastighet (2 längdenheter per ms) med synapsens längd. Vid denna framtida tidpunkt schemaläggs ett anrop av synapsens kör-funktion. När kör-funktionen anropas sker följande: det schemaläggs en körning av målneuronen vid impulsens sluttid (efter 2 ms), synapsen lagrar nuvarande tid i medlemsvariabeln ”lastSpikeArrival”, och plasticitetsfunktionen ”synapticPlasticity” anropas. Denna funktion förändrar synapsens styrka, vilket är en faktor när impulsstyrkan beräknas. Med andra ord har en stark synaps en större påverkan på målneuronen. En excitatorisk synaps har enklare att förmedla en impuls vidare, och en stark inhibitorisk synaps (med ett negativt styrkevärde), har enklare att förhindra att målneuronen avfyras. Plasticitetsfunktionen (som implementerar STDP-inlärning) anropas, som tidigare omnämnt, vid två tillfällen: när målneuronen avfyras och när en signal når fram till målneuronen i synapsen. Funktionen utgår från att det lämnas ett spår direkt efter en avfyrning i alla neuroner respektive synapser. Detta spår får värdet 1 vid avfyrning, och avtar sedan exponentiellt över tid (65 % per ms i neuroner, 75 % avtagande per ms i synapser). Beräkningen av värdet av de två spårvariablerna gör genom: $S_n = a_n{}^{t_{nu}-t_n}$, samt $S_s = a_s{}^{t_{nu}-t_s}$, där S är spår-variabler, a är avtagande per ms ($a_n = 0.65$, $a_s = 0.75$), t_{nu} är nuvarande tid, t_s är tid när en signal senast nådde fram till målneuronen i synapsen (”lastSpikeArrival”), och t_n är tid vid början av senast impuls i målneuron. Det är viktigt att spår-variablerna uppdateras efter att plasticitetsfunktionen har anropats för att inlärningen ska fungera som avsett. Detta garanteras genom att låta variabeln bli 0.0 och den var exakt lika med 1.0. När spårvariablerna är beräknade används de för att beräkna den resulterande skillnaden i styrka av synapsen genom $\Delta w = f_g * (f_s * S_s - f_n * S_n)$, där f_g är den globala inlärningsfaktorn (med standardvärde 1.0) som används för att kontrollera hur hastigt synapsstyrkorna förändras globalt. Variablerna f_s (standardvärde 0.13) och f_n (standardvärde 0.30) ändrar plasticitetsfunktionens preferens för negativa jämfört med positiva Δw. Efter att skillnaden i synapsstyrka har applicerats så används värde-clamp för att hålla styrkan inom ett visst omfång (0.0 till 1.0 för excitatoriska synapser, -1.0 till 0.0 för inhibitoriska).

2.4. Användning

2.4.1. Bibliotek

En användare av biblioteket importerar min kod i dennas projekt, och kan på så sätt använda min simulationsmodell utan att själv behöva förstå hur simulationen fungerar i detalj. En förståelse av de externa gränssnitt jag har designat är istället tillräckligt. Koden kan klonas med GitHub

(länk till projekt finns under Bilaga 1). Kloningen kan ske genom standard konsolbaserad GitHub, även om jag rekommenderar att använda GitHub Desktop då det har ett grafiskt användargränssnitt som är enklare att använda. Efter detta krävs att användaren går igenom den process att installera nödvändiga bibliotek, som anges i avsnitt 2.1 Bygga programmeringsmiljö. För att hantera simulationen krävs endast att användaren hanterar objekt av klassen ”NeuCor”. De publika funktioner och variabler som finns i denna klass är dokumenterade i header-filen på engelska, och på svenska nedan.

```
NeuCor(int n_neurons);
// Konstruerar klassen. n_neurons parameter är antal neuroner som skapas initialt

void run();
// Kör hela simulationen

float runSpeed;
// Hur mycket tid (i ms) som simuleras när run() anropas (standardvärde 1.0)

bool runAll;
// Sant falskt värde om huruvida alla neuroner bör simuleras (standardvärde
falskt)

float getTime() const;
// Returnerar hur mycket simulationstid (i ms) som har förflutit

float learningRate;
// Är den globala inlärningsfaktorn fg (standardvärde 0.6)

float presynapticTraceDecay, postsynapticTraceDecay;
// Avtagande per ms av alla synapsers respektive neuroners spårvariabler
(standardvärde 0.5, samt 0.9)

void setInputRateArray(float inputs[], unsigned inputCount, coord3
inputPositions[] = {NULL}, float inputRadius[] = {NULL}));
// Parametern ”inputs”  är en array av float-värden som anger (i Hz)
avfyrningsfrekvensen av de avfyrare som finns i hjärnan. ”inputCount” anger
antalet element i denna array. Eftersom att det är minnesadressen till array som
förvaras, så uppdateras avfyrarna automatiskt efter de värden dom finns på
minnesadressen när run() anropas. Parametrarna ”inputPositions” och ”inputRadius”
är ej nödvändigt, men definierar positionerna och radien på avfyrarna i hjärnan.

void addInputOffset(unsigned inputID, float t);
// Förskjuter given avfyrares (inputID) nästa avfyrning med given tid t (ms). Går
även att använda negativa tidsvärden.


void createNeuron(coord3 position);
// Skapas positioner vid givna koordinater. Om NAN anges som koordinatvärden, blir
positionen slumpmässig.

void createSynapse(std::size_t toID, std::size_t fromID, float weight);
// Skapar synaps mellan från en neuron (”toID”) till en annan (”fromID”) med given
styrka (”weight”).

void makeConnections();
// Skapar två kopplingar mellan alla neuroner mindre än 1 längdenhet från
varandra.
```

Kodavsnitt 5 Visar användning och funktion av de publika medlemmarna i ”NeuCor”-klassen.

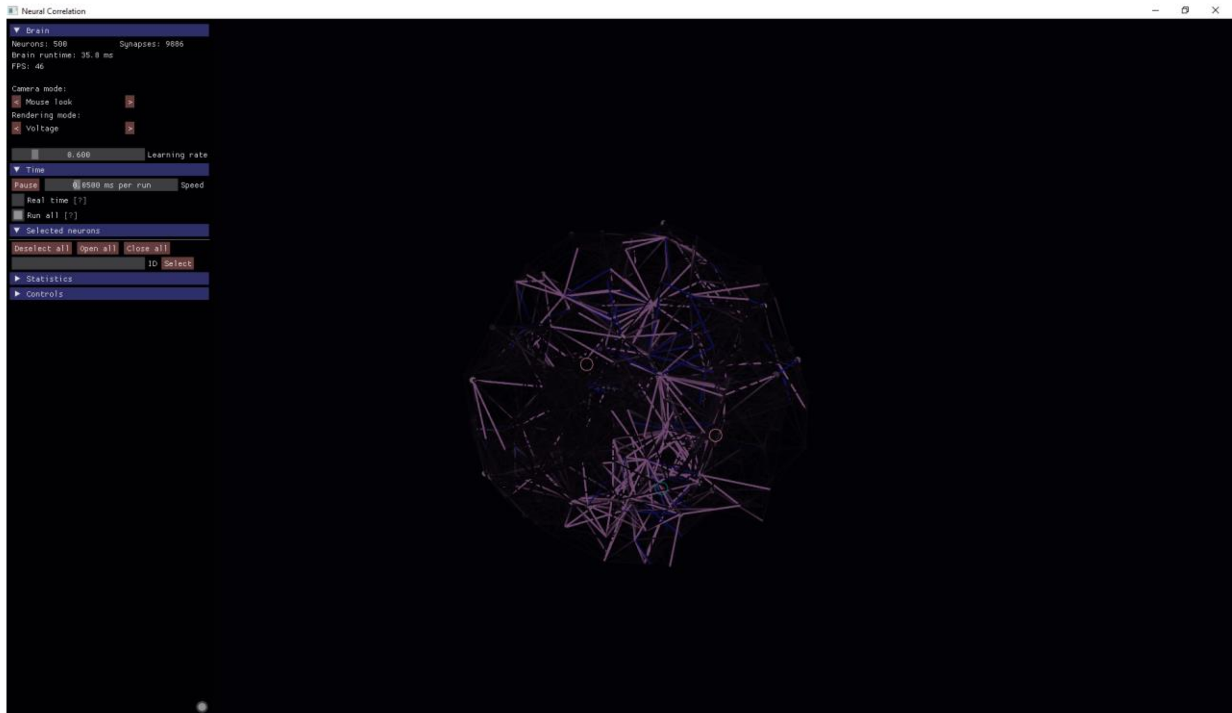
Renderaren har fler publika medlemsvariabler och funktioner. Av dessa kommer dock endast de nödvändiga förklaras då flera av dessa antingen är självförklarande, eller är till för att genom extern kod kunna ändra status på sådana variabler som annars går att ändra på inom användargränssnittet. Dokumentation för dessa essentiella medlemmar av klassen ”NeuCor_Renderer” står nedan. Ett exempelprogram som demonstrerar hur ”NeuCor” och ”NeuCor_Renderer” kan användas tillsammans finns i Bilaga 3.

```
NeuCor_Renderer(NeuCor* _brain);
// Konsturerar renderare. “_brain” parameter är en pointer till hjärnan.

float getDeltaTime();
// Returnerar verklig tid som har passerat sedan körning av updateView()


bool runBrainOnUpdate;
// Om renderaren bör köra hjärnan i när updateView() (standardvärde falskt)

void updateView();
// Renderar hjärnan och visar den i fönstret

void pollWindow();
// Ber Windows ange de handlingar som användaren har utfört. Denna funktion är
nödvändig för att Windows inte ska registrera fönstret som fruset.
```

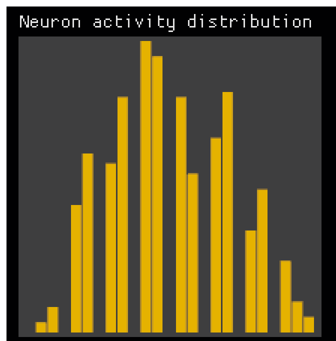
Kodavsnitt 6 Visar användning och funktion av publika medlemmar i ”NeuCor_Renderer”-klassen. Bygga programmeringsmiljö

2.4.2. Grafiskt gränssnitt

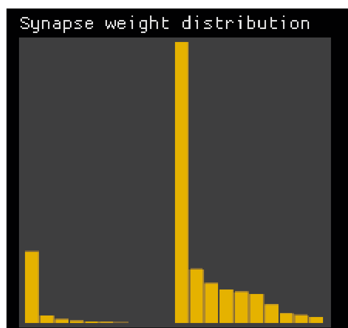
En förkompilerad version av koden finns att ladda ner under Bilaga 2. Det grafiska gränssnittet består av dels renderingen av hjärnan, och dels de paneler som biblioteket ImGui används för att skapa. Med detta gränssnitt går det att avläsa information om hjärnan för att förstå dens beteende, och till viss del även att ändra hur hjärnan körs. När renderaren först öppnar fönstret (se Figur 6) går det att använda och flytta runt menyerna. Genom att klicka på rubriken ”Controls” så öppnas en lista som detaljerar hur man kan använda tangentbord och mus för att manipulera renderingen. Mätning kommer bland annat göras med de tre grafer som finns under menyn ”Statistics” (se Figur 7, Figur 8, Figur 9), neuron-potentialsgrafen (som visas när en neuron markeras och dess fönster öppnas, se Figur 10), och synaps-styregrafen (som finns under synapsmenyer i neuron-fönstret, se Figur 11). På grund av begränsningar med ImGui har det ej varit möjligt gradera dessa grafer.



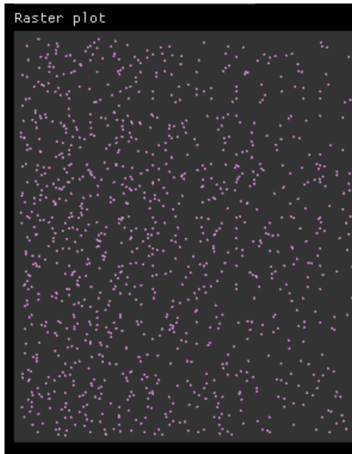
Figur 6 Visar ett exempel på hur rendering av paneler (vänster) samt hjärnan ser ut på skärmen. Synapserna visas som streck, vita ifyllda ringar är neuroner, icke ifyllda ringar är avfyrare.



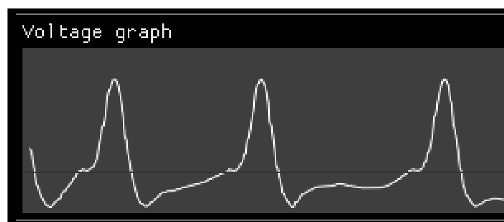
Figur 7 Visar gränssnittsgraf (under menyn "Statistics") som indikerar fördelningen av avfyrningsfrekvenser (x-axel) av neuroner i hjärnan. Grafens egenskaper kan ändras genom att klicka på den.



Figur 8 Visar gränssnittsgraf (under menyn "Statistics") som indikerar fördelningen av avfyrningsfrekvenser (x-axel) av neuroner i hjärnan. Grafens egenskaper kan ändras genom att klicka på grafen.

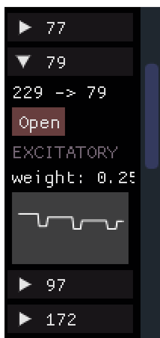


Figur 9 Visar rastergraf (under menyn "Statistics") av neuroners avfyrningar. Varje prick är avfyrning. X-axel är tid (närmast i tid till vänster), y-axel är neuronens ID-nummer. X-axelns omfång går att ändra genom att klicka på grafen.

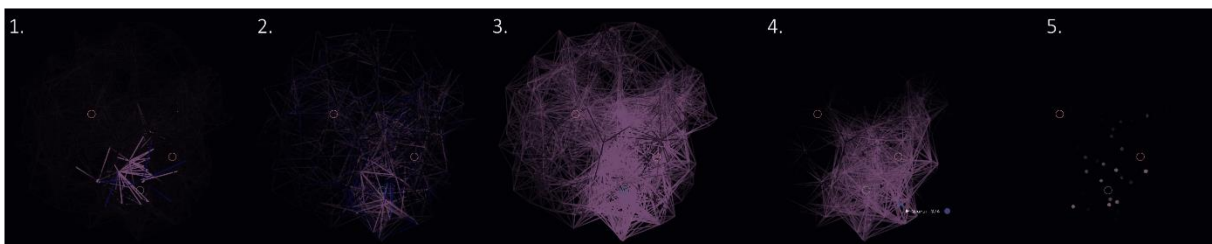


.

Figur 10 Visar potentialgraf (neuron-fönster) av vald neurons spänning (y-axel) över tid (x-axel, nämasti i tid till vänster). Den mörka linjen markerar tröskelpotential.



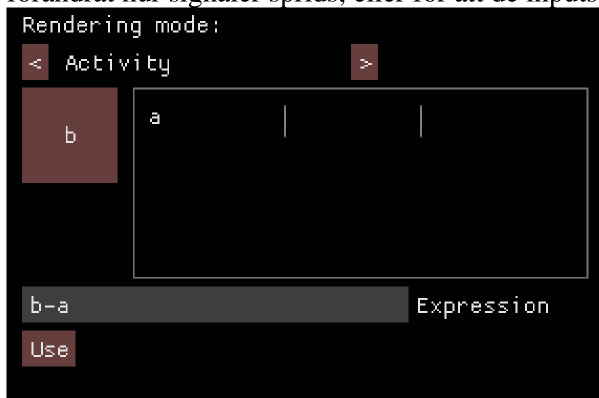
Figur 11 Visar synapsinformation (utsynaps-listan längs ner i neuron-fönster), för synapsen som går från neuronen med id 229 till målneuronen med id 79. Grafen visar synapsstyrka (x-axel, närmast i tid till vänster) över tid.



Figur 12 Visar de olika renderingslägena. Läge 1 visar spänning av synapser samt synapsernas typ (rosa är excitatoriska, blå är inhibitoriska). Läge 2 visar synaptisk styrka (rosa är excitatoriska, blå är inhibitoriska) med avfyrningssekvens som faktor (går att stänga av i meny) för att de viktigaste synapserna ska synas tydligast. Läge 3 visar avfyrningsfrekvens av presynaptisk/postsynaptisk neuron (beroende på ände av synapslinje). Läge 4 visar hur signalerna sprider sig från de markerade neuronerna. Detta tar hänsyn till synapsstyrka. Läge 5 renderar endast neuroner.

Utöver dessa grafer finns det även ett antal olika renderingslägen som presenterar information i renderingen genom att ändra genomskinligheten av synapserna. De fem olika renderingslägena förklaras i Figur 12. Renderingsläget som visar avfyrningssekvenser har utökad funktionalitet (Figur 13), som är användbart för analys då det illustrerar hur aktiverade vissa delar av hjärnan är i olika

tillstånd. Förändring av aktivitetsmönster i nätverket kan antingen ske på grund av att plasticiteten har förändrat hur signaler sprids, eller för att de inputs som kommer genom avfyrarna har förändrats.

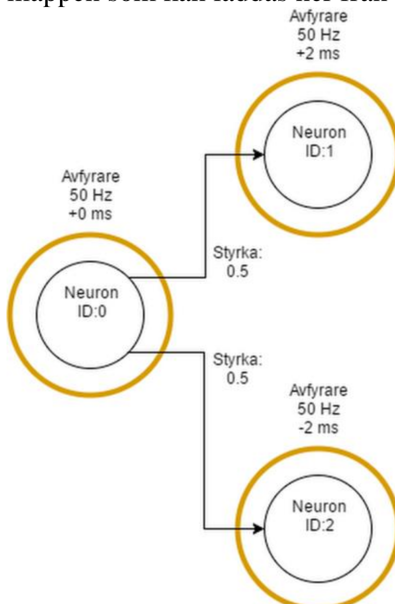


Figur 13 Visar menyn för renderingsläget som visar avfyrningsfrekvenser. Nuvarande värden kan sparas i variabler (bokstäverna) genom att klicka på den stora röda knappen. Dessa värden kan sedan bearbetas genom det aritmetiska uttryck som anges i textfältet, vars resultat visas i renderingen av synapserna. Det är även möjligt att återställa neuronernas aktivitet (Ctrl och klick på stora röda knappen), så ett nytt genomsnitt kan beräknas.

2.5. Resultat

2.5.1. 3 neuroner med 3 avfyrare

3 neuroner placeras och kopplas ihop enligt Figur 14. Varje neuron har en avfyrare var, med avfyrningsfrekvensen 50 Hz. Frekvenserna för avfyrarna för neuronerna med ID 1 och 2, är förskjutna med +2, respektive -2 ms. Detta innebär att neuron 1 alltid kommer avfyra 2 ms efter neuron 0, och neuron 2 alltid kommer avfyra 2 ms före neuron 0. Alla neuroner simuleras vid varje körning, och resten av inställningarna har standardvärden. Programmet har namnet ”3_neurons_3_inputs.exe” i mappen som kan laddas ner från länken i Bilaga 2.



Figur 14 Neuronen med ID 0 kopplar till neuronerna 1 och 2 (ensriktat) med synapser av styrkan 0.5. Varje neuron har varsin avfyrare med frekvensen 50 Hz. Impulserna från avfyrarna för neuronerna 1 och 2 är förskjutna med +2, samt -2 ms.

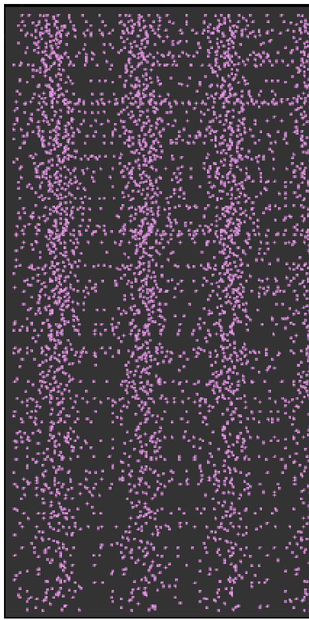
Denna konfiguration resulterar i att synapsen från neuron 0 till 1 succesivt ökar tills styrkan 1.0 är nådd, och att synapsen från neuron 0 till 2 succesivt minskar tills styrkan 0.0 är nådd. Detta tar i snitt ca 120 ms simulationstid. På grund av bakgrundsaktivitet aktiveras neuronerna ibland slumpmässigt, vilket påverkar synapsstyrkorna, även om det i slutändan inte förändrar det slutgiltiga tillståndet.

2.5.2. 750 neuroner med 1 avfyrare

Här placeras en avfyrare på koordinaterna (2.0, 0.0, 0.0), med radien 0.8 och en konstant avfyrningsfrekvens av 35 Hz, i en hjärna med 750 neuroner. Alla andra inställningar har

standardvärden. Programmet har namnet ”750_neurons_1_input.exe”. De slumpmässiga variablerna utgår från nuvarande tid, och kommer av den anledningen att variera från körning till körning. Därför presenteras i denna resultatdel nätverkets normala uppförande, även om en specifik körnings uppförande kan variera.

I början av simulationen är aktiviteten i hjärnan endast utlöst av bakgrundsavfyrningarna. Efter ca 15 ms har dock en exponentiell fortplantning av impulserna initierats, som på kort tid aktiverar större delen av neuronerna i nätverket. Under detta stadie förskjuts majoriteten av de positiva synapsstyrkorna mot 0 (vilket kan avläsas över tid från grafen i Figur 8). Neuronernas aktivitet minskar även under ett förlopp av ca 160 ms. Vid denna tidpunkt fortplantar sig avfyrarens impulser endast ett kort avstånd innan de dör ut. Efter 5 sekunders simulationstid sprider sig impulserna från avfyraren till resten av hjärnan. Det uppstår även mindre områden av konstant aktivitet (se Figur 15), som sedan dör ut.

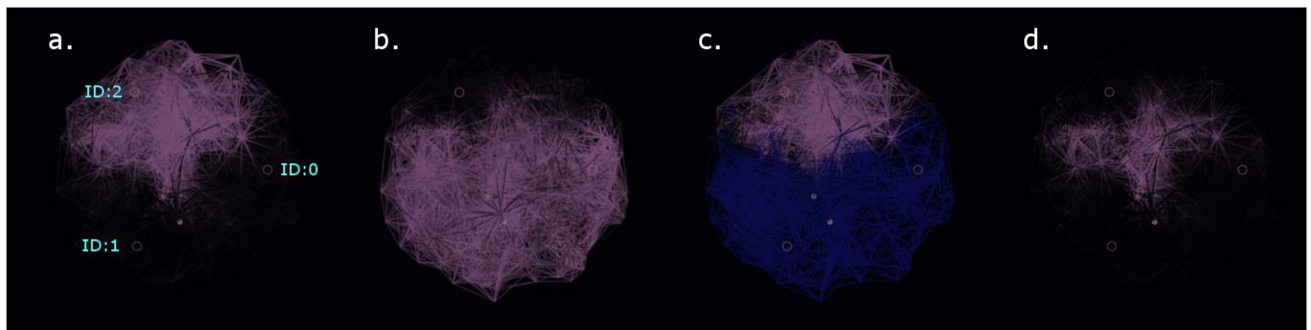


Figur 15 Rastergraf efter 5 sekunders simulationstid av nätverk med en avfyrare. Det periodiska mönstret över tid uppstår när avfyraren (35 Hz) skickar impulser. De horisontella raderna visar neuroner som är del av mindre självbärande kretsar, och vars aktivitet därför är konstant.

2.5.3. 750 neuroner med 3 avfyrare

Detta test har samma konfiguration som det innan, med annat antal och ett annat beteende av avfyrarna. Programmet heter ”750_neurons_3_inputs.exe”. Tre avfyrare med radien 0.8 placeras i en liksidig triangel med ett avstånd av 2 från centrum av nätverket (origo). Avfyrare 0 och 1 har alltid samma avfyrningsfrekvens. De initiala avfyrningsfrekvenserna för avfyrare 0 och 1, samt 2, är ett slumpmässigt värde mellan 0 och 75 Hz. Varje millisekund uppdateras avfyrningsfrekvenserna genom att addera ett slumpmässigt värde mellan -1 och 1, men kan inte gå under eller över värdena 0 respektive 75 Hz. Vid 10.0 sekunder påbörjas mätningarna. Inlärningsfaktorn (f_g i avsnitt 2.3 Implementering) blir 0, vilket stänger av inlärningen så att mätningarna inte påverkar hjärnans beteende. Under perioden 10.0 till 10.2 sekunder blir alla avfyrningssekvenser 0, och mellan perioden 10.2 och 10.6 sekunder får endast avfyrare 2 frekvensen 50 Hz. Mellan 10.6 till 10.8 sekunder slås åter igen alla avfyrare av, och mellan perioden 10.8 och 11.2 sekunder så får avfyrare 0 och 2 frekvensen 50 Hz.

Även här övergår nätverket efter ca 15 ms till global konstant aktivitet, vilket får synapsstyrkorna att sjunka. Resultatet av mätningarna efter 10 sekunder redovisas nedan.



Figur 16 (a.) Visar hjärnans aktivitet under perioden 10.2 till 10.6 sekunder, då endast avfyraren med ID 2 är aktiv (50 Hz). Avfyrarnas ID i förhållande till position gäller även för resten av figuren. (b.) Visar hjärnans aktivitet under perioden 10.8 till 11.2 sekunder, då avfyrarna 0 och 1 är aktiva (båda 50 Hz). (c.) Visar hjärnans aktivitet i a. subtraherat med hjärnans aktivitet i b (verktyget i Figur 13 används). Här syns alltså de neuroner som avfyrades under endast perioden i a. som rosa, och de endast under perioden i b. som blå. (d.) Visar aktiviteten i a. multiplicerat med aktiviteten i b. Detta visar de områden som är aktiva under båda perioderna.

3. Analys och diskussion

Tanken bakom simulationskonfigurationen i avsnitt 2.5.1 var att till viss del efterlikna det verkliga experimentet gjort i rapporten ”Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type”. [8] Här har två fysiska neuroner med en synaps mellan stimulerats regelbundet och synkroniserat, med ett visst intervall tidsskillnad. Ett positivt tidsintervall, då den presynaptiska neuronen avfyras efter den postsynaptiska neuronen, resulterar i en gradvis ökning av den potential som synapsen orsakar i postsynaptiska neuronen, och ett negativt tidsintervall ger istället en minskning. I simulationen görs samma sak, även om det för enkelhetens skull är synapsstyrkan som mäts direkt istället för resulterande potential. Resultatet visar att denna simulationsmodell stämmer överens med verkligheten (inom ramarna av detta test). Detta betyder dock inte att alla aspekter av plasticiteten modelleras. Till exempel så beror i verkligheten förändringen i synapsstyrka även på själva synapsstyrkan. En svag synaps har enklare att bli starkare än en redan stark synaps.[8] Jag har valt att inte inkludera detta för att inte komplicera analysen samt för att inte göra det svårare att balansera hjärnans aktivitetsnivå. De inhibitoriska synapsernas uppförande är inte baserat på någon verklig studie, utan följer samma regler som de excitatoriska synapserna. Allmänna och enkla regler som alltid gäller är inom programmering en bra taktik för att undvika buggar, och bidrar till ett robustare system som ofta också är enklare att förstå. Detta resulterar dock i ett dilemma när man modellerar biologiska system. Evolutionen har en förmåga och en tendens att konstruera system med en hög komplexitet, som därmed är svåra att modellera med exakthet. Därför krävs det att jag som programmerare dels gör bedömningen till vilken detaljnivå simuleringen bör modellera, samt vilka delar av de biologiska systemen som går att förenkla. Att använda ett kontinuerligt system för synapsstyrkorna, i stället för att simulera olika typer och mängder av neurotransmittorer är en sådan förenkling. Den digitala domänen ger ofta friare spelregler än den fysiska, vilket innebär att digitala system ofta inte kräver samma komplexitet som de biologiska för att uppnå samma funktionalitet. Det finns dock andra begräsningar med de digitala systemen, så som datorkraft, som man som programmerare behöver ta hänsyn till.

Simulationskonfigurationen i 2.5.2 visar att nätverket själv kan motverka den återkoppling av impulser som till en början orsakar den kaosartade konstanta aktiviteten i hela hjärnan. Det är kaoset som gör det omöjligt för en synaps att konsekvent avfyras före dess målneuron, vilket gör att alla synapser försvagas. Det går även att se hur impulserna från avfyraren fortplantar sig till allt större delar av nätverket. Trots att det alltid går en synaps i vardera riktning, så betyder STDP-plasticiteten att bara en av dessa synapser kommer bli stark, samtidigt som den andra blir svag. Detta innebär att avfyrarens impulser enkelriktas ut från själva avfyraren. Beslutet att lägga till bakgrundsaktivitet kom sent i arbetet, men visade sig vara nyckeln till att få det område som avfyrarnas impulser fortplantar sig till

[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.

att expandera över tid. Bakgrundsavfyrningar i inaktiva eller redan aktiva delar av nätverket påverkar sällan synapsstyrkorna. Om dock en bakgrundsavfyrning sker i en inaktiv neuron vid gränsen av avfyrarens aktiva område så sker två möjliga saker beroende på om bakgrundsavfyrningen sker direkt före eller efter att avfyrarens impuls anländer. Om den sker direkt före så försvagas synapsen mellan den aktiva och den inaktiva synapsen något, vilket i sin tur inte leder till något. Om den sker direkt efter så stärks synapsen, vilket i sin tur kan leda till att den aktiva neuronen själv kan avfyra den tidigare inaktiva neuronen, vilket leder till att synapsen stärks ytterligare. Nettoeffekten av detta är att det är mer troligt att bakgrundsaktiviteten leder till att avfyrarens aktiva område blir större än mindre. Dock så uppstår det områden av bara ett fåtal neuroner, som alltid är aktiva. Detta för att signalerna återkopplar, och därmed blir självbärande. Både formandet och destruktionen av dessa kretsar verkar triggas av bakgrundsaktiviteten, men hur de fungerar och kan motverkas behövs undersökas vidare. En ide är att införa ett energisystem, som ”utmattar” annars evigt aktiva neuroner.

Det sista experimentet, i avsnitt 2.5.3, har syftet att testa frågeställningen: kan systemet bygga upp en intern intuitiv modell av sambanden mellan dess inputvärden? Hjärnan ges tre inputs (genom avfyrarna), varav två är länkade (avfyras alltid samtidigt). Det faktum att de två avfyrarna som är länkade, kopplas ihop och isoleras från den icke länkande avfyraren (Figur 16), visar att hjärnan har hittat och förstår sambandet. Detta bekräftar frågeställningen, även om det inte säger mycket om i vilken utsträckning hjärnan faktiskt kan göra detta. För att mäta detta skulle det till exempel gå att öka antalet inputs. Skulle hjärnan reda ut sambanden mellan 10 avfyrare? Det går även att komplicera typen av samband mellan input-värdena, genom att till exempel fördröja tiden mellan att två avfyrare aktiveras, eller genom att utnyttja den temporala kodning som systemet tillåter. I princip så bör detta nätverk kunna finna en lång rad av olika sorters orsakssamband. Genom att mata in pixelvärden från en bild som inputs, så skulle nätverket sannolikt lära sig klassificera mellan olika objekt i bilden. Ett hinder för detta är dock beräkningsintensitet, som ökar skarpt med antal neuroner. En mängd beräkningsmässiga optimeringar skulle kunna utföras, även om det ibland skulle vara på bekostnad av den biologiska modelleringens verklighetstrogenhet. Att implementera ett sorts belöningssystem skulle vara ett givet nästa steg för detta projekt, då detta skulle ge möjligheten att låta hjärnan styra virtuella agenter för att lösa andra problem än att hitta samband. En annan möjlig riktning är att träna upp en sorts allmänbildning hos en given hjärna, vilket var den ursprungliga avsikten med arbetet. Detta system skulle då bli ett sätt att låta datorer själva utveckla en grundläggande intuitiv förståelse av koncept som fysik och psykologi.

4. Källförteckning

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. Bilagor

Bilaga 1 - GitHub projektlänk
https://github.com/Axelwickm/NeuroCorrelation/tree/Snapshot

Bilaga 2 - Förkomplilerade exempel nerladdning
https://drive.google.com/drive/folders/0B56HcM6Y9mY5VkNWQnY4S0xCRms?usp=sharing

Bilaga 3 - Ett enkelt program som först skapar en hjärna med 100 neuroner, och sedan öppnar ett fönster som visar när denna hjärna renderas.

```
#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;
}
```