Cognon Neural Model Software Verification and Hardware Implementation Design

by

Pau Haro Negre

B.S., Escola Tècnica Superior d'Enginyeria de Telecomunicacions de Barcelona - BarcelonaTech (UPC), 2011

B.S., Facultat d'Informàtica de Barcelona - BarcelonaTech (UPC), 2011

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical, Computer, and Energy Engineering

2013

This thesis entitled:
Cognon Neural Model Software Verification
and Hardware Implementation Design
written by Pau Haro Negre
has been approved for the Department of Electrical, Computer, and Energy Engineering

_____

Albin J. Gasiewski

_____

Frank Barnes

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Haro Negre, Pau (M.S., Electrical Engineering)

Cognon Neural Model Software Verification
and Hardware Implementation Design

Thesis directed by Prof. Albin J. Gasiewski

Little is known yet about how the brain can recognize arbitrary sensory patterns within milliseconds using neural spikes to communicate information between neurons. In a typical brain there are several layers of neurons, with each neuron axon connecting to $\sim 10^4$ synapses of neurons in an adjacent layer. The information necessary for cognition is contained in theses synapses, which strengthen during the learning phase in response to newly presented spike patterns.

Continuing on the model proposed in [24], this study seeks to understand cognition from an information theoretic perspective and develop potential models for artificial implementation of cognition based on neuronal models. To do so we focus on the mathematical properties and limitations of spike-based cognition consistent with existing neurological observations. We validate the cognon model through software simulation and develop concepts for an optical hardware implementation of a network of artificial neural cognons.

Dedication

To my sister, my parents, and Maria.

Acknowledgements

I would like to thank Pete J. Balsells, the Generalitat de Catalunya, and the University of Colorado at Boulder for their support through the Balsells Graduate Fellowship Program at the University of Colorado.

I am very grateful to Professor Albin J. Gasiewski for his advice, knowledge, and support during these last two years. I am also thankful to the other thesis defense committee members, Professor Frank Barnes and Professor David Beeman for their educative and inspiring observations.

I would also like to thank all my friends in the University of Colorado and Boulder who have accompanied me all over these last two years.

Finally, a very special thanks to my sister, my parents, Maria, and my whole family, for their unconditional support and encouragement.

Contents

Chapter

1 Introduction 1

1.1 Background 1

1.1.1 The Human Brain Project 1

1.1.2 SpiNNaker 2

1.1.3 SyNAPSE 3

1.1.4 BrainScaleS 3

1.1.5 Numenta 4

1.2 Purpose of the Study 5

1.3 Overview of the Thesis 5

2 Information Processing Within Mammalian Brain 6

2.1 Neocortex 7

2.2 Nerve Cells 10

3 Neural Cognon Model 12

3.1 Cognon Basic (CB) Neuron Model 12

3.1.1 Basic Recognition 13

3.1.2 Neural Learning Model 17

3.2 Cognon Extended (CE) Model 18

3.2.1 Synpase Atrophy (SA) Learning Model 19

3.2.2 Feedback Paths 20
3.2.3 Innovation and Relevance 21
3.3 Software Simulation of the Family of Cognon Models 21
3.3.1 Design 22
3.3.2 Implementation 23
3.3.3 Results 23

4 Hardware Architectures 28
4.1 Spatial Phase Modulator (SPM) 28
4.1.1 Description 30
4.1.2 Learning 32
4.1.3 Performance 35
4.2 Spatial Amplitude Modulator (SAM) 36
4.2.1 Description 36
4.2.2 Learning 41
4.2.3 Performance 43

5 Conclusions 44
5.1 Suggestions of Future Research 45

Bibliography 46

Appendix

A Source code 48
A.1 cognon_basic.py 48
A.2 test_cognon_basic.py 51
A.3 cognon_extended.py 53

A.4 test_cognon_extended.py 58

A.5 run_experiment.py 64

A.6 test_run_experiment.py 68

A.7 create_tables.py 71

Tables

Table

2.1 Number of cortical neurons in mammals 9

3.1 CB model parameters 14

3.2 CE model parameters 19

3.3 False alarm recognition probabilities of the CB model 26

3.4 Learning and false alarm probabilities of the CB model 26

3.5 Values of L (bits/neuron) for the CE model 27

4.1 Lens properties 39

4.2 LCD properties 40

4.3 Diffuser properties 40

4.4 Detector properties 41

4.5 LED properties 41

Figures

Figure

2.1 Three drawings of cortical lamination 8

2.2 Neuron diagram 10

3.1 Cognon Basic (CB) neuron model architecture 13

3.2 General cognon model architecture 15

3.3 Two neurons in a single layer of a dual-flow network 20

4.1 3D render of the SPM architecture 29

4.2 Diagram of the SPM implementation 30

4.3 Gerchberg-Saxton algorithm diagram 34

4.4 3D render of the SAM architecture 37

4.5 Diagram of the SAM implementation 38

Chapter 1

Introduction

Currently it is a challenging intellectual problem to understand how the human brain computes so rapidly using neural spikes. Despite the enormous amount already known about the brain's biochemistry, structure and connections, little is known about how the computation works.

1.1 Background

In recent years, projects based upon massive hardware simulation of interconnected neurons have been undertaken with the goal of achieving a physical validation of a biologically inspired model neural network. For example, the Spiking Neural Network Architecture (SpiNNaker) project is developing a computer architecture to simulate the human brain by using parallel ARM processors and a self-timed Network-on-Chip (NoC) communications system [8]. Another project, Blue Brain, aims at building large-scale computer simulations of neuronal microcircuits with high biological fidelity that run on the 360 teraFLOPS IBM Blue Gene/L supercomputer [18]. This section includes a description of the most relevant projects that are currently in active development.

1.1.1 The Human Brain Project

The Human Brain Project (HBP), which integrates the Blue Brain project, is a research project that aims to develop a large-scale information and communications technology (ICT) infrastructure for the specific purpose of understanding the brain and its diseases, and of translating this knowledge into new computing technology. The end hopes of the HBP include being able to

mimic the human brain using computers and being able to better diagnose different brain problems. The project is directed by the École Polytechnique Fédérale de Lausanne (EPFL) and co-directed by Heidelberg University, the University Hospital of Lausanne and the University of Lausanne. As of November 2011 the largest simulations were of mesocircuits containing around 1 million neurons and 1 billion synapses. A full-scale human brain simulation of 86 billion neurons is targeted for 2023.

In medicine, the project's results will facilitate better diagnosis, combined with disease and drug simulation. In computing, new techniques of interactive supercomputing, driven by the needs of brain simulation, will impact a range of industries, while devices and systems, modelled after the brain, will overcome fundamental limits on the energy-efficiency, reliability and programmability of current technologies, clearing the road for systems with brain-like intelligence.

1.1.2 SpiNNaker

SpiNNaker is a million-core massively-parallel computing platform designed as a collaboration between several universities and industrial partners, led by Steve Furber at the University of Manchester, and inspired by the working of the human brain [9]. The project flagship goal is to be able to simulate the behaviour of aggregates of up to a billion neurons in real time.

Each SpiNNaker chip embeds 18 ARM968 energy-efficient processors. The chip has six bidirectional, inter-chip links that allow networks of various topologies, controlled by the Network-on-Chip subsystem. The design is based on a six-layer thalamocortical model developed by Eugene Izhikevich [13]. The largest SpiNNaker machine will contain 1,200 Printed Circuit Board (PCB) with 48 SpiNNaker nodes mounted PCB. It will be capable of simulation a 10^9 simple neurons, or $\sim 10^6$ of neurons with complex structure and internal dynamics. In operation, the engine will consume at most 90 kW of electrical power.

1.1.3 SyNAPSE

Systems of Neuromorphic Adaptive Plastic Scalable Electronics (SyNAPSE) is a Defense Advanced Research Projects Agency (DARPA) program with the goal to develop electronic neuromorphic machine technology that scales to biological levels. The ultimate aim is to build an electronic microprocessor system that matches a mammalian brain in function, size, and power consumption. It should recreate 10^{10} neurons, 10^{14} synapses, consume 1 kW (same as a small electric heater), and occupy less than 2 l of space.

The project is currently in the third phase out of five. In the first phase cortical simulations were developed with $\sim 10^9$ neurons and $\sim 10^{13}$ synapses running on the Blue Gene/P supercomputer, which had 147,456 CPUs and 144 terabytes of memory [1]. The second phase produced a 45 nm CMOS neuromorphic chip that implements 256 leaky integrate-and-fire neurons and 1,024 synapses per neuron with a power consumption of 45 pJ per spike [21]. As the third phase is completed, it is expected that multi-core neurosynaptic chips with $\sim 10^6$ neurons per chip will be announced this year.

1.1.4 BrainScaleS

Brain-inspired multiscale computation in neuromorphic hybrid systems (BrainScaleS) is a collaboration project of 18 research groups from 10 European countries with the aim to understand function and interaction of multiple spatial and temporal scales in brain information processing. The project focuses on three research areas: in vivo biological experimentation; simulation on petascale supercomputers; and construction of neuromorphic processors. Their goal is to extract generic theoretical principles to enable an artificial synthesis of cortical-like cognitive skills.

The BrainScaleS project has already fabricated a neuromorphic hardware based on wafer-scale analog VLSI. Each 20-cm-diameter silicon wafer contains 384 chips, each of which implements 128,000 synapses and up to 512 spiking neurons. This gives a total of ~200,000 neurons and 49 million synapses per wafer. The circuitry contains a mix of both analog and digital circuits, as the

simulated neurons themselves are analog, while the synaptic weights and interchip communication is digital.

1.1.5 Numenta

Numenta is a company that uses a technology named Grok that is based on biologically inspired machine learning technology to build solutions that help companies automatically and intelligently act on their data. The company was founded by Jeff Hawkins, who previously founded the Redwood Neuroscience Institute, a scientific institute focused on understanding how the neocortex processes information. In 2004 he wrote the book On Intelligence [12], which describes the progress on understanding the neocortex.

The core model of Grok technology is the Cortical Learning Algorithm (CLA), which is a realistic model of a layer of cells in the neocortex. This model is based on the idea that the neocortex is not a computing system, it is a memory system. In the CLA, the brain is described as a predictive modeling system, where learning is achieved by building models of the world from streams of sensory input. From these models, predictions are made, anomalies detected, and actions taken.

The three principles behind the CLA are:

- In contrast with how computers store information, data stored in the brain is very sparse, only a few percent of the neurons in the brain are active at a given time. A data storage system named Sparse Distributed Representations (SDRs) is presented in this model, where active neurons are represented by 1s and inactive neurons are 0s. SDRs have thousands of bits, but typically only about 2% are 1s and 98% are 0s.

- Most machine learning techniques assume that each data point is statistically independent of the previous and next records, but the brain uses the information of how patterns change over time to make predictions. The primary function in the neocortex is sequence memory. Learning sequences allow the brain to predict what will happen next.

- On-line learning allows the brain to keep up with the amount of new information that enters through sensory inputs, by continuously refining the internal models of the world. When sensory data enters the brain there is no time to store it and process it later. Every new input needs to be processed, and learned if it is relevant.

1.2 Purpose of the Study

Part of the problem in understanding animal cognition consists of knowing how highly interconnected networks of biological neurons can rapidly learn, store and recognize patterns using neural spikes. Numerous models have been proposed to explain this [14], but few have progressed to a stage of hardware implementation while retaining the essential biological character of neurons.

The purpose of this work is to develop new computation architectures that implement a model of neurons and their interconnections, based on the model proposed in [24]. The objectives of this research project are thus to extend the models presented in the book, and both simulate in software and implement in hardware a network of modeled neurons of a size comparable to, e.g., small insects such as ants. Such a demonstration would allow us to determine if the cognon model can provide new insights into understanding how the brain works, by joining multiple cognon models of neurons, forming simple cognon neural networks, and studying how they interact to achieve the cognition results observed experimentally in the brain of small animals to humans.

1.3 Overview of the Thesis

The thesis is organized as follows: Chapter 2 contains a brief introduction of the human brain structure and characteristics, focusing on the neocortex and the neurons in this region. Chapter 3 contains the description of the cognon model, both the basic recognition and learning model and the cognon extended model. Numerical results and intercomparison with the original results are provided and discussed in this chapter. Chapter 4 contains the description of the two proposed hardware architectures to implement the cognon model. Chapter 5 contains the conclusions and future work direction.

Chapter 2

Information Processing Within Mammalian Brain

The mammalian brain is the most complex organ of the body and the main part of the central nervous system. Specifically, the central nervous system is a bilateral and essentially symmetrical structure with seven main parts [15]:

(1) The **spinal cord**, the most caudal part of the central nervous system, receives and processes sensory information from the skin, joints, and muscles of the limbs and trunk and controls movement of the limbs and the trunk.

(2) The **medulla oblongata**, which lies directly above the spinal cord, includes several centers responsible for vital autonomic functions, such as digestion, breathing, and the control of heart rate.

(3) The **pons**, which lies above the medulla, conveys information about movement from the cerebral hemisphere to the cerebellum.

(4) The **cerebellum** lies behind the pons and is connected to the brain stem by several major fibres tracts called peduncles. The cerebellum modulates the force and range of movement and is involved in the learning of motor skills.

(5) The **midbrain**, which lies rostral to the pons, controls many sensory and motor functions, including eye movement and the coordination of visual and auditory reflexes.

(6) The **diencephalon** lies rostral to the midbrain and contains two structures. One, the **thalamus**, processes most of the information reaching the cerebral cortex from the rest of the central nervous system. The other, the **hypothalamus**, regulates autonomic, endocrine, and visceral function.

(7) The **cerebral hemispheres** consist of a heavily wrinkled outer layer (the **cerebral cortex**) and three deep-lying structures: the **basal ganglia**, the **hippocampus**, and the **amygdaloid nuclei**. The basal ganglia participate in regulating motor performance; the hippocampus is involved with aspects of memory storage; and the amygdaloid nuclei coordinate the autonomic and endocrine responses of emotional states. The cerebral cortex is divided into four lobes: frontal, parietal, temporal, and occipital.

Despite the mammalian brain consists of a number of network structures that are critical for higher level cognition, in this work we focus on the cerebral cortex, as it is the primary part of the central nervous system involved in higher intelligence and cognition.

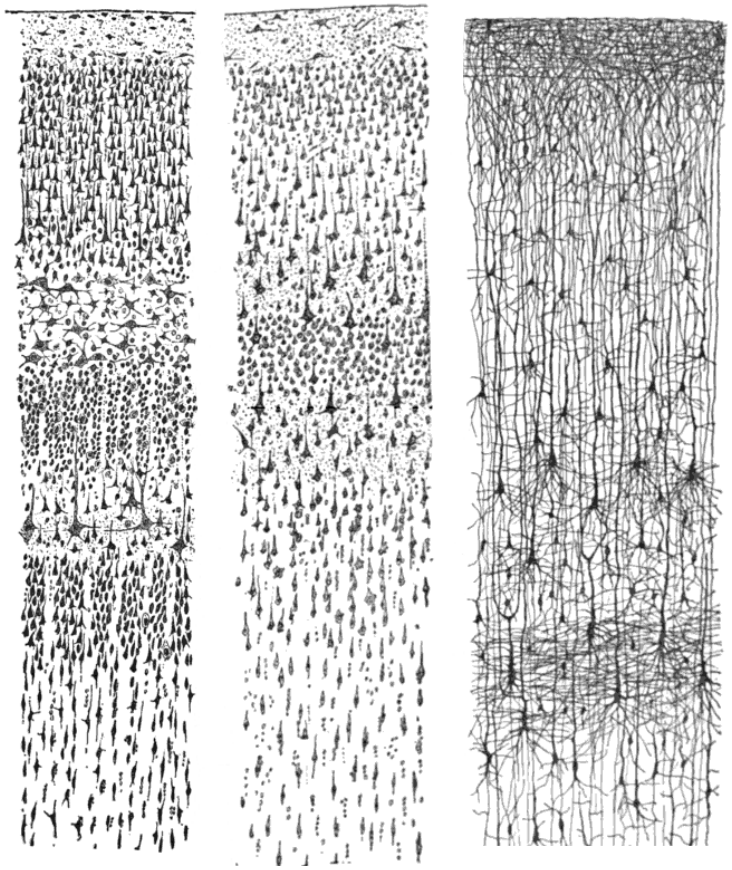
2.1 Neocortex

Almost everything we think of as intelligence (perception, language, imagination, mathematics, art, music, and planning) occurs primarily in the neocortex, which is the most phylogenetically recent structure within the cerebral cortex.

The neocortex is a $\sim$2 mm thick wrinkled sheet of grey matter that wraps around the white matter that wires the various cortical regions together. Although both cortical neuron characteristics and their connection patterns vary from region to region, most cortex has six visibly defined layers of physically differentiated neurons. Neurons in various layers connect vertically to form small highly interconnected cascading networks called columns. This structure exhibits relatively few visible differences across all cortex. Figure 2.1 shows three drawings of cortical layers developed by Santiago Ramon y Cajal where this structure can be observed.

Stretched flat, the human neocortical sheet is roughly the size of a large dinner napkin. Other

Figure 2.1: Three drawings of cortical lamination by Santiago Ramon y Cajal [4], each showing a vertical cross-section, with the surface of the cortex at the top. Left: Nissl-stained visual cortex of a human adult. Middle: Nissl-stained motor cortex of a human adult. Right: Golgi-stained cortex of a $1^1/_2$ month old infant. The Nissl stain shows the cell bodies of neurons; the Golgi stain shows the dendrites and axons of a random subset of neurons. The six different cortical layers can be identified.



mammals have smaller cortical sheets: the rat's is the size of a postage stamp; the monkey's is about the size of a business-letter envelope [12]. But regardless of size, most of them contain six layers similar to what you see in a stack of business cards [12].

The neocortex is basically made of a dense network of tightly packed nerve cells. Currently the exact amount, or density, of neurons is not know precisely. Continuing with the stack of business cards equivalent, a square of one millimeter on a side on the top of the stack marks the position of an estimated 10^5 neurons. Estimates have been developed during the years for different mammals cerebral cortex. Some of these values are shown in Table 2.1. The human cerebral cortex is estimated to have in average 19 billion neurons in female brains and 23 billion in male brains, a 16% difference [22].

Table 2.1: Number of cortical neurons in mammals [23, 22].

Animal name	Number of cortical neurons (in millions)
Mouse	4
Dog	160
Cat	300
Horse	1,200
African elephant	11,000
Human	19,000 - 23,000

Naked eye observation of the neocortex presents almost no landmarks, the convoluted surface looks very similar in all parts. However, there are are two clear characteristics: a giant fissure separating the two cerebral hemispheres, and the prominent sulcus that divides the back and front regions.

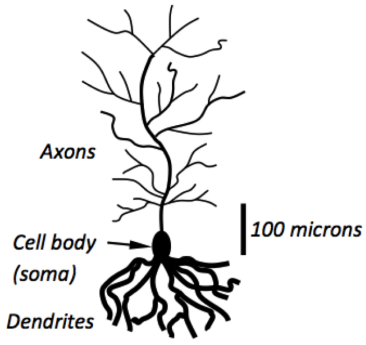
Even though the appearance of the neocortex is homogeneous, neuroscientists have been able to associate some mental functions to certain regions. Different functional regions perform visual, auditory, tactile, somatic (smell), motor, memory, and other functions. For example, perception of anything on the left side of the body and the space around it is located close to the right parietal lobe. Another example can be found in the left frontal region known as Broca's area, where an

injury may compromise the ability to use the rules of grammar, although the vocabulary and the ability to understand the meanings of words are unchanged.

2.2 Nerve Cells

Nerve cells, or neurons, are the main signaling units of the nervous system. They are the basic computational units of the brain and perform both logic and wiring functions as single interconnected cells. A typical neuron has four morphologically defined regions: the cell body or soma, dendrites, the axon, and presynaptic terminals. Each of these regions has a distinct role in the generation of signals between nerve cells. Figure 2.2 shows a representation of a typical neuron.

Figure 2.2: Typical neuron with its dendrite arbor below, the soma or cell body in the center, and the axon arbor above. Neuronal signals propagate from axons across synapses to dendrites in potential spikes of ∼1 millisecond duration [24].



The **cell body** is the metabolic center of the cell. It contains the nucleus, which stores the genes of the cell, as well as the endoplasmic reticulum, an extension of the nucelus where the cell's proteins are synthesized. **Dendrites** branch out from the soma in tree-like fashion and are the main apparatus for receiving incoming signals from other nerve cells. In contrast, the **axon** extends away from the cell body and is the main conducting unit for carrying signals to other neurons. An axon can convey electrical signals along distances ranging from 0.1 mm to 3 m. These electrical signals, called **action potentials**, are rapid, transient, all-or-none nerve impulses, with an amplitude of 100 mV and a duration of about 1 ms.

When the axon from one neuron touches the dendrite of another, they form small connections called **synapses**. Synapses are where the nerve impulse from one cell influences the behavior of another cell. A neural signal, or spike, arriving at a synapse can make it more likely for the recipient cell to spike. Some synapses have the opposite effect, making it less likely the recipient cell will spike. Thus synapses can be inhibitory or excitatory. The exchange of a synapse can change depending on the behavior of the two cells.

Observations of the brain show that neural signals consist mostly of spikes of perhaps one-millisecond duration, or bursts of such spikes. Spike-based processing present in neurons can support rapid learning and recognition of complex patterns. A spike from a typical neuron might stimulate 10,000 other neurons, each of which might then feed 10,000 others.

Chapter 3

Neural Cognon Model

The cognon model is an information-based model of neural spike computation and cognition presented by David H. Staelin and Carl H. Staelin [24]. It is based on well-known properties of cortical neurons. In it, a layered spike-processing neural model is presented that is arguably consistent with both observed neural behavior and sub-second learning and recall. Fast learning is explained using a feed-forward scheme, which permits to train multiple layers of neurons sequentially in approximately linear time. Some extensions to the model are considered, which include spike timing, dendrite compartments, and new learning mechanisms in addition to spike-timing-dependent plasticity (STDP). The model is verified using the Shannon information metric (recallable bits/neuron) to measure the amount of information taught to a model neuron using spike patterns, both in theory and simulations.

3.1 Cognon Basic (CB) Neuron Model

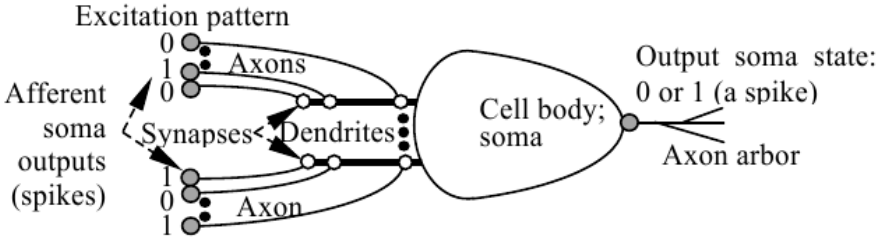
The Cognon Basic (CB) model is based in a well-known characteristic of neurons: they produce an output spike when the sum of nearly simultaneous (within a few milliseconds) identical input spikes weighted by each synapse strength exceeds some firing threshold H that varies with time, where the set of simultaneously excited neural inputs defines the neuron input pattern [20, 16, 25]. The CB model assumes all spikes have the same amplitude and shape, while synapses have only two possible strengths, 1 and G ($G > 1$). Also, this model does not learn and respond to time intervals between the successive arrival times of spikes at individual synapses, because this is an

unclear mechanism and adequate neuron model performance is obtained using this simpler model.

3.1.1 Basic Recognition

The architecture of the single-neuron computational model is represented in Figure 3.1. The excitation pattern is defined as the set of output spikes of all the afferent neurons, which are the neurons that have axons connected to the current neuron, at the soma output before the axons. This model differs from common neuron definitions in that the axons from previous neurons are included as part of the current neuron, and the axon of this neuron is excluded. For this reason the output of the neuron is considered at the output of the cell body, before the axon arbor.

Figure 3.1: Diagram of the Cognon Basic (CB) neural model architecture and soma-based definition of excitation patterns [24].



Each excitation pattern is a sequence of 0's and 1's that correspond respectively to the absence or presence of a spike at the afferent soma (input neurons) within the same nominal millisecond window during which spikes can superimpose efficiently. As displayed in Figure 3.1, the pattern is not the input to the synapses, which could differ if the paths between the various afferent soma and the summing point introduce additional different fixed delays. The output (0 or 1) is defined by whether the neuron produces nearly simultaneous spike at the neuron output (right-hand side of the figure), where each output spike signifies recognition of the input as familiar.

Neuron output spikes propagate along a highly branched axon arbor that might extend to ∼0.5 mm length or more. Each small axon branch typically terminates in a synapse that connects to a dendrite on another neuron. Dendrites are generally shorter and thicker than axons and connect

to the cell body and soma where the basic neuron model sums the excitations over a sliding time window that can be approximated using discrete time intervals of ∼1 millisecond width. If the sum equals or exceeds the firing threshold, then the basic neuron model fires. In nature, patterns might be presented at intervals of tens of milliseconds, and perhaps at the well-known gamma, theta, or other periodicities of the brain [24].

Figure 3.2 displays the main parameters of the general cognon architecture. Each layer of neurons has a set of binary outputs defined as $\overline{s}_j$ where j identifies the layer index. In a layer, each cognon-modeled neuron is identified by the index i. If we consider the modeled neuron i of layer j, its output is defined as s_{ij}, and the set of synapse weights is defined as $\overline{a}_{ij}$.

This figure also defines the different states of a synapse. It can be **active** or **not active**, depending on whether the previous layer $j-1$ neuron i axon output $s_{i,j-1}$ is connected to one of the dendrites of the current layer j neuron i forming a synapse. An active synapse can be **excited** when the previous layer neuron connected to the synapse fires, and so the output is 1. Finally, in the CE model an extension to the basic model is considered where some synapses may be **atrophied** and even later removed if these synapses are found not to be useful to learn the patterns which the neuron is exposed to.

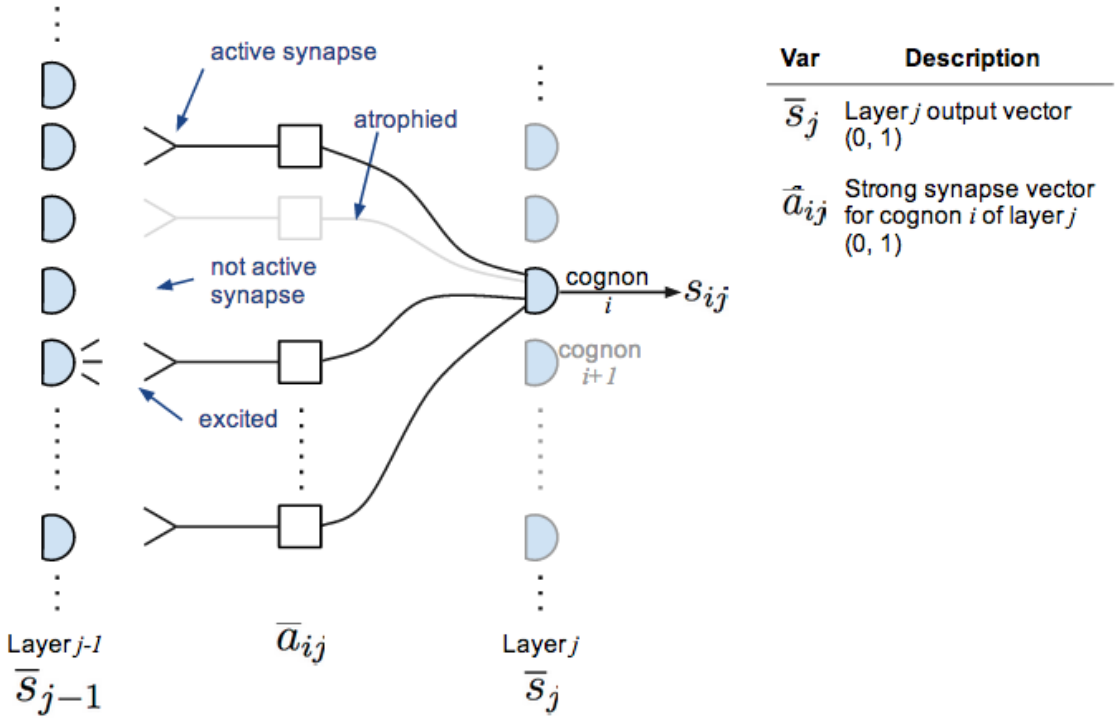
Table 3.1: CB model parameter description and approximate values.

Parameter	Description	Value*
S_0	Number of synapses per neuron	∼10,000
N	Number of excited synapses at the neural model input	
w	Number of patterns taught while a neuron is learning ready	$w \lesssim S_0/3N$
H	Firing threshold	$9 < H < 60$
G	Ratio of strong synapse strength to weak synapse strength	$1.3 < G < 1.8$
L	Recallable learned Shannon information	
$\overline{s}_j$	Layer j output vector	0 or 1
$\overline{a}_{ij}$	Strong synapse vector for cognon i of layer j	0 or 1
$\overline{\overline{P}}_{ij}$	Projection matrix of the active synapses for cognon i of layer j	0 or 1

*Approximate values and ranges in real neurons.

Based on the parameters presented in Table 3.1 the model can be described formally as

Figure 3.2: Diagram of the general cognon model architecture.



follows. Synapses for cognon i of layer j are connected to a subset of the previous layer $j-1$ axons to obtain the active excited synapses:

$$\overline{\overline{P}}_{ij}\overline{s}_{j-1} \tag{3.1}$$

The number of strengthened active synapses that are excited at a certain time can be expressed as:

$$\overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} \tag{3.2}$$

In a similar form, the number of not strengthened active synapses that are excited at that same time can be described as:

$$\overline{s}_{j-1}^{t}\overline{\overline{P}}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} - \overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} \tag{3.3}$$

Using these definitions, cognon neuron i fires when the number of excited strengthened synapses weighted by G plus the number of non-strengthened but excited synapses is higher than the threshold defined by GH:

$$\left(\overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1}\right)G + \left(\overline{s}_{j-1}^{t}\overline{\overline{P}}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} - \overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1}\right)1 > GH \tag{3.4}$$

Which can also be expressed as:

$$\overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} > \frac{GH - \overline{s}_{j-1}^{t}\overline{\overline{P}}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1}}{G-1} \tag{3.5}$$

In the CB model, the number of excited synapses at the neural model input N is fixed. Thus, we can simplify using the equality:

$$\overline{s}_{j-1}^{t}\overline{\overline{P}}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} = N \tag{3.6}$$

And the condition deduced previously can be reduced to:

$$\overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} > \frac{GH - N}{G - 1} \tag{3.7}$$

Finally, if the number of excited synapses N is close to the number of synapses needed to fire a neuron H, the previous expression can be further simplified:

$$\overline{a}_{ij}^{t}\overline{\overline{P}}_{ij}\overline{s}_{j-1} \gtrsim H \quad (N \simeq H) \tag{3.8}$$

As later will be shown in the simulation results section, the basic recognition neuron model can instantly recognize even hundreds of patterns with little error. This is also true of the early Willshaw model [11] and a few others, so instant recognition alone is not unique to this model. The uniqueness lies instead in the relative neurological plausibility of the fast learning mechanism proposed for training the basic recognition neural model.

3.1.2 Neural Learning Model

Once the CB model has been shown to be able to recognize known patterns by using the embedded information in the input synapse strength (1 or G), this section describes how this basic neural recognition model could learn new patterns or determine which ones to learn.

The proposed learning model is based on a simplified form of spike-timing-dependent plasticity (STDP) for which any spike that arrives in a timely way so as to help trigger an output spike instantly strengthens its synapse from a weight of unity to G ($G > 1$). Since this basic model is binary, no other synapse strengths are allowed. The authors designate this as the synapse-strength (SS) learning model.

The main SS model learning mechanism is that when a learning-ready neuron is being trained, any pattern that excites an output spike also irreversibly strengthens the weights of all contributing afferent synapses from 1 to G. This assumption is based on neurological observations which show that spike-triggered dendritic back-propagation is known to strengthen afferent synapses [19].

In contrast with most basic neural models, the SS model has the property of instant learning.

This permits to avoid the mathematical NP-complete back-propagation training time barrier, where neuron learning times increase roughly exponentially with the number of synapses per neuron, because learning is accomplished almost instantly within a single neuron rather than requiring to consider the interaction between pairs of neurons. This roughly assumes that any pattern presented to a neuron while it is learning ready, or plastic, merits memorization.

In order to be able to evaluate the SS model, a metric to measure the information stored per neuron is defined as the learned Shannon information L (bits/neuron) recoverable from a binary neuron [2]:

$$L \simeq w \left[(1 - p_L) \log_2 \frac{1 - p_L}{1 - p_F} + p_L \log_2 \frac{p_L}{p_F} \right] \tag{3.9}$$

Where w is the number of patterns, p_L is the probability of a neuron learning a pattern presented to it, and p_F is the probability of false alarm, which is the probability of firing triggered by unlearned patterns.

This Shannon metric applies when: 1) the desired information is the taught information that can be recovered by observing the model's outputs for all possible input excitation patterns, and 2) the only information provided by a neural spike is that the excitation pattern responsible for that spike had excited approximately the same synapses that had been strengthened earlier as a result of seeing similar excitation patterns when the synapses were plastic and learning-ready. In contrast with other previous models, which often consider that some information may reside in the time interval between two consecutive spikes, no other learned-information storage and recovery mechanism is assumed for the basic neuron model.

3.2 Cognon Extended (CE) Model

The Cognon Extended (CE) model extends the basic cognon model by introducing the optional possibilities that: 1) neuron firing decisions might be performed within dendritic sectors (compartments) that independently sum their own synapse excitations and test that sum against

their own firing threshold before firing the neuron, and 2) the relative timing of spikes within a single neuron excitation pattern (which might last 2-20 milliseconds) could further distinguish one pattern from another. An additional learning model involving synapse atrophy (SA) is also considered, in which synapses that not contribute are atrophied and replaced with potentially more useful ones.

Table 3.2: CE model parameters definition.

Parameter	Description
C	Number of dendrite compartments capable of firing independently
D	Number of possible time slots where neurons can produce spikes
S_0	Number of synapses per neuron
N	Number of excited synapses at the neural model input
w	Number of patterns taught while a neuron is learning ready
H	Firing threshold
G	Ratio of strong synapse strength to weak synapse strength
L	Recallable learned Shannon information
p_F	False alarm probability for a random excitation pattern
p_L	Probability that a given taught pattern will be learned by a neuron
R	Refractory period

3.2.1 Synpase Atrophy (SA) Learning Model

The SA learning model is a new learning mechanism proposed in the CE model. It can be combined with the previously presented SS model or work alone. The main advantage of this learning model is that, combined with the SS model, it can help reduce the false alarm probability that increases when synapses with reduced weight (unity) still contribute to the sum which is tested against the threshold $G \cdot H$ and can therefore cause the neuron to fire erroneously.

In order to reduce this problem, the SA learning model proposes to atrophy synapses that never contributed to a spike during learning, despite many opportunities. Less useful synapses should be replaced with potentially more useful ones linked to other neurons, thus maintaining the total number of synapses per neuron roughly constant over most of each neuron's life. As neurons

maturate from young states, the number of synapses typically increases with time and then slowly declines as the neuron becomes old, perhaps after yeards or decades.

3.2.2 Feedback Paths

Rich feedback between layers of cognon model neurons might permit improved noise immunity, learning and recognition of pattern sequences, compression of data, associative or content-addressable memory functions for both static and time-sequential patterns, and development of communication links through white matter. It is well known that in many cortical areas there are more feedback synapses conveying information top-down from higher levels than there are synapses conveying information bottom-up from sense such as the auditory system [12].

Figure 3.3: Two cognon modelled artificial neurons (N) in a single layer of a dual-flow network having top-down feedback paths. Set A synapses are feed-fordward and Set B synpases convey feedback [24].

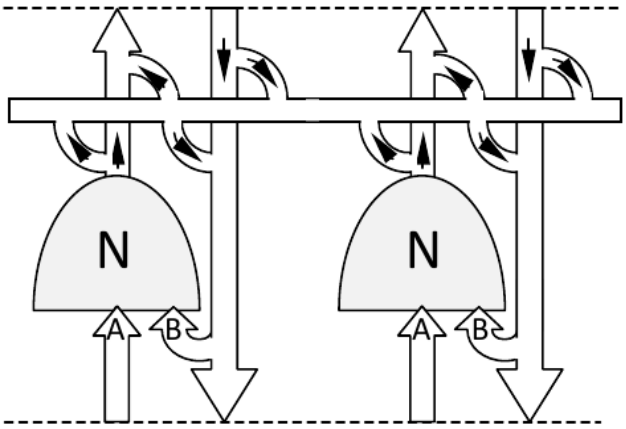


Figure 3.3 illustrates how the cognon neuron model might accommodate and rapidly train synapses handling such bidirectional flows of information. This would allow to implement the equivalent of a full-duple communications system. The afferent synapses for each neuron can be divided into the “A” bottom-up set that accepts spikes from lower sensory layers, and the “B” top-down set that accepts spikes from the outputs of that neural layer and any layer above.

3.2.3 Innovation and Relevance

No prior models are arguably consistent with both observed neural behavior and sub-second learning and recall. Also, an analysis considering the information stored in the neuron as learnt patterns was not done previously.

Many neuroscientists ask how simplified numerical models of neurons could contribute useful understanding of cognition since they differ so markedly from real neurons. The behavior of **good** simple numerical models is a subset of the achievable complex behaviours of real neurons, however they can establish an approximate lower bound to real performance.

3.3 Software Simulation of the Family of Cognon Models

Starting from the models and algorithms presented in [24], we developed a software to simulate the CB and CE models. The development is available at https://github.com/pauh/neuron for download and study.

The cognon models family simulator trains and tests an ensemble of neurons having given parameters and reports the average results for various parameters, particularly the mean and standard deviation for the probability of learning p_L, probability of false alarm p_F, and learned information metric L.

To obtain sufficiently accurate statistics for the probability of learning, the system trains enough neurons so that at least 10,000 words are exposed to neurons during training. At least ten neurons are trained, no matter how many words are in the training set for each neuron. Regarding the probability of false alarm, each neuron configuration is tested on a total of at least 1,000,000 random words that do not belong to the training set. Each neuron instance is tested on at least 1,000 random words.

3.3.1 Design

The design of the software system is based on object-oriented design, where a set of interacting abstract objects are defined to solve the problem. The system is structured in two modules: the cognon model, and the auxiliary classes to run different setups.

The cognon model module contains the central class `Neuron`, which represents a CE model neuron. It stores the neuron parameters ($S0$, H, G, C, $D1$, and $D2$) and the state of each synapse (strength, delay, and container). It can also be used to model a CB neuron, by setting the parameters C, $D1$, and $D2$ to 1, as the CE model is an extension of the CB model. In addition to the main class, three other classes are defined to help with the neuron training process.

The `Synapse` class represents a connection between the neuron's input dendrites and the output axons of other neurons. It contains the offset of the represented synapse in the list of neuron synapses, and the associated delay, which represents the time the signal takes to traverse the axon to reach the synapse. An excitation pattern is represented using the `Word` class, which contains the input synapses that fired for a given time period. Finally, the `WordSet` class stores a time sequence of `Word` instances that are used to train and test a neuron.

The module that contains the auxiliary classes to run experiments with the cognon model neuron is divided in four main classes. The first two classes (`Alice` and `Bob`) are based on a classic metaphorical example from communications theory: suppose that Alice wishes to tell Bob something that Bob should know, and that she does this by means of a single new neuron that she trains. She trains the neuron while it is plastic by exposing it to a set of w patterns she chooses from all possible patterns. Bob then extracts that maximum information by exposing the neuron to all possible patterns and noting which patterns produce recognition signatures and are therefore presumed to have been selected and taught by Alice. Using these two classes a neuron can be trained with `Alice` class, and the learning probability p_L and false alarm probability p_F can be obtained from the `Bob` class instance.

Two other classes are included in this module. The `Configuration` class is used to represent

a set of parameters to run a simulation experiment. The `Cognon` class is the main class of the simulation system, it runs a set of experiments as described in the given configuration and processes the obtained results.

3.3.2 Implementation

The design was implemented using Python programming language and the numerical libraries NumPy. It was developed in a standard Linux environment, but should also work on any other environment where Python and NumPy are available. In order to verify that the implementation is coherent with the design unit tests were developed for each class in the system. Multiprocessing capabilities of Python were used to run independent experiments at the same time when multiple computer processors are available.

A first implementation of the most basic neural model was developed initially, which can be found in the `cognon_basic.py` source code file. This version only includes the `Word` class and a simplified version of the `Neuron` class. It allows to test the first example in the book, which can be verified using the accompanying script file `test_cognon_basic.py` that includes tests for both classes. Within the neuron training tests, the `test_train()` method is defined, which runs the book example and makes sure that the result is the expected one.

Learning from the experience of this first prototype, the final design was developed and implemented. The main requirements considered were clarity of the implementation and code to facilitate reusability, and optimization of the resulting program where this requirement did not conflict with the clarity of the solution.

3.3.3 Results

In order to verify that the results from the book and the implemented solution were the same, simulations were run with the parameters shown in the book results tables.

Table 3.3 presents the average false alarm probabilities p_F that result for the set of parameter values proposed in the cognon book. The definition of each parameter can be found in Table 3.1.

These simulations use the most basic model presented previously in this section. Also, the number of excited inputs N is fixed in each simulation.

The low values for p_F near 1 percent suggest that in these cases the CB neuron model should emit no more than one spontaneous erroneous spike every hundred patterns or so. It can be observed that p_F is very sensitive to both N and w, by comparing the first and second rows and first and third rows respectively.

The main result from Table 3.3 is that the CB neuron model can instantly recognize hundreds of patterns with little error. In addition to instant recognition, the fast learning mechanism of this model is a unique feature not found in other models.

The top part of Table 3.3 shows that the acceptable false alarm probability p_F limits both N and w. It also shows how larger neurons (larger S_0) can store many patterns before failing due to an excessive number of learned patterns w, which is equivalent to over education. The lower part of Table 3.3 shows that lower values of G, even close to 1, are enough to keep p_F low, and this is more consistent with synaptic values observed in neurons.

Table 3.4 shows the simulation results of the full CB model. In this case, the number of excited inputs N is not fixed, instead the probability that any given synapse is excited for any particular excitation pattern is $1/R$ and is independent and identically distributted among synapses and patterns, following a binomial distribution.

The values of S_0 and H are fixed, but the values of R, w, and G were extracted from an optimization to maximize the L. The resulting p_F and p_L estimates have been obtained averaging the results of many experiments.

Table 3.5 presents the simulation results for the full CE model. In this case the modelled neurons are tested with different numbers of independent dendrite compartments C and more than one available inter-pattern delay D. The simulations are for neurons with 10,000, 1,000, and 200 synapses.

In the results it can be observed that having variable delays enable neurons to learn more information. However, this software implementation does not work as expected for multiple com-

partments. The neurons seem to be over-trained, as the probability of learning p_L is 1.0, and the probability of false alarm p_F is also 1.0, all patterns are being recognized even if the neuron was not trained with these patterns. The simulations of the original authors did not expose this problem, so further inspection of the development might show how to overcome this unexpected results.

Table 3.3: False alarm recognition probabilities p_F as a function of the basic recognition model neuron parameters with a fix number N of excited synapses.

$p_F(\%)$	N	H	S_0	w	G	L	L/S_0
0.47	4	4	10	1	100	7.7	0.77
10.22	5	4	10	1	100	3.3	0.33
14.70	4	4	10	2	100	6.6	0.66
0.00	10	10	100	4	100	6.3	0.06
0.02	11	10	100	4	100	26.9	0.27
0.11	11	10	100	5	100	50.7	0.51
0.46	11	10	1,000	60	100	467.7	0.47
0.44	11	10	10,000	600	100	4730.8	0.47
0.35	22	20	10,000	450	100	3682.4	0.37
0.03	10	10	100	6	1.5	56.4	0.56
0.00	11	10	1,000	15	1.5	31.9	0.03
0.01	11	10	10,000	160	1.5	603.5	0.06
1.51	14	10	10,000	10	1.5	60.7	0.01

Table 3.4: Values of L, p_F, and p_L as a function of the CB model parameters when the number N of excited synapses is binomially distributed about its mean S_0/R.

L	$p_F(\%)$	$p_L(\%)$	H	G	S_0	R	w
759.4	1.19	72.4	30	4.0	10,000	303	200
574.0	0.09	85.4	105	4.0	10,000	86	70
413.9	0.14	53.2	40	1.9	10,000	250	100
181.4	0.80	18.3	5	3.6	1,000	333	300
119.8	0.69	40.3	10	3.6	1,000	111	60
115.9	1.92	18.4	5	1.9	1,000	333	300
107.3	0.40	56.3	15	4.0	1,000	66	30
34.7	1.23	25.9	5	3.6	200	57	40
26.6	1.73	58.0	10	4.0	200	20	10
9.5	0.42	20.5	20	1.9	200	12	10

Table 3.5: Values of L (bits/neuron) as a function of the CE model parameters, based on the maximized values in [24].

C	D	S_0	L	H	R	G	$p_F(\%)$	$p_L(\%)$	w
10	4	10,000	0	5	125	1.8	100.00	100	2000
1	4	10,000	1117	5	384	3.8	1.15	58	400
4	4	10,000	155	5	178	3.2	43.62	76	500
10	1	10,000	0	5	333	3.8	100.00	100	200
4	1	10,000	2	10	357	3.6	99.48	100	300
1	1	10,000	756	30	303	4.0	1.23	72	200
1	1	1,000	163	5	285	4.0	1.52	27	200
4	4	1,000	4	5	25	1.9	96.36	99	200
1	4	1,000	158	5	83	1.9	1.14	13	500
4	1	1,000	10	5	83	3.8	88.55	100	60
10	4	1,000	0	5	10	1.8	100.00	100	70
1	1	200	34	5	57	3.8	1.71	28	40
1	4	200	31	5	16	1.8	1.09	14	80
4	1	200	2	5	16	3.8	87.07	100	10
4	4	200	1	5	5	1.9	96.80	99	40

Chapter 4

Hardware Architectures

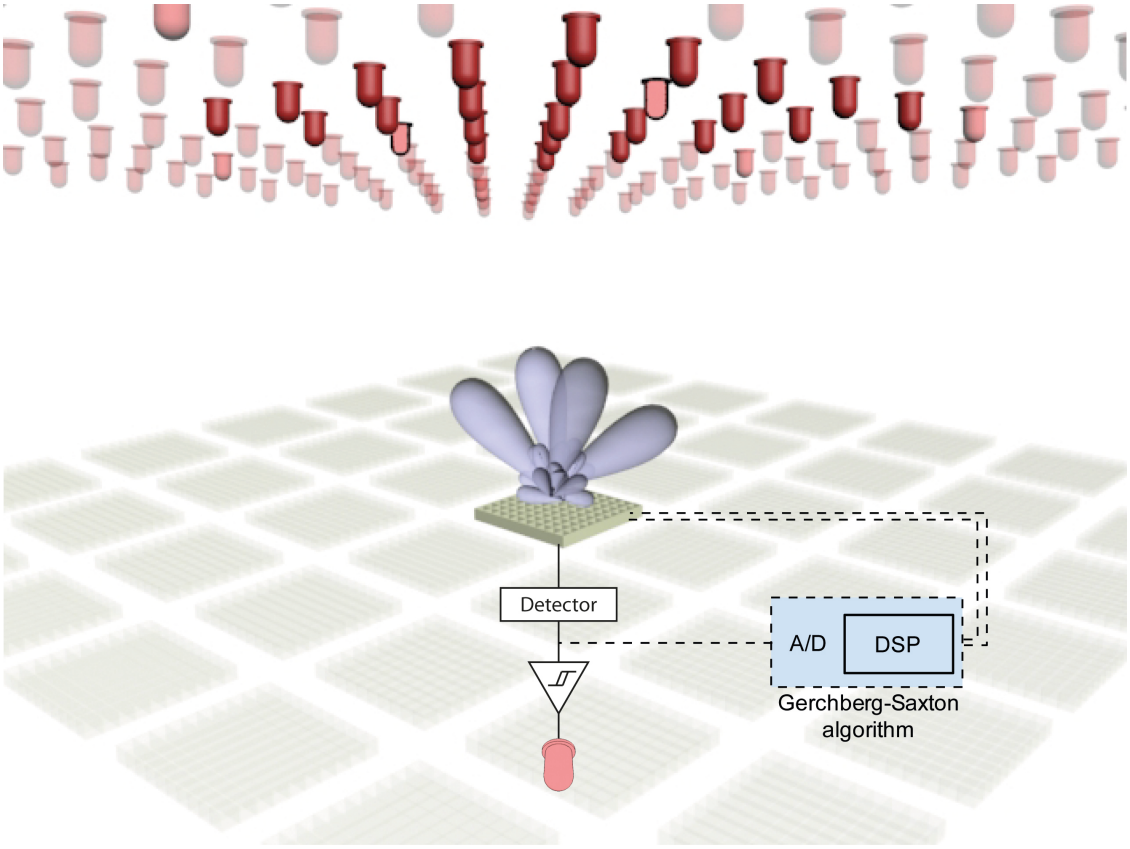
In this chapter two new hardware architectures are presented. These are inspired on the information-based model of neural spike computation and cognition presented by David H. Staelin and Carl H. Staelin, the cognon model [24]. Both architectures are based on optical devices, and aim to build a set of layers of cognons. Connections are built between each contiguous layer to simulate the synapses between neurons.

Based on the cognon model simulations from the previous chapter and the devices used in both approaches, a performance in learning speed for each cognon of less than one second is expected. However, the recognition speeds might be much faster, as it involves only optical devices.

4.1 Spatial Phase Modulator (SPM)

The first proposed architecture implementation is based on using spatial light modulators (SLMs) to create beams that simulate the synapses between neurons, and being able to modify the weights of these synapses by changing the properties of the generated beams. This approach can easily implement the plasticity that has been observed in the synapses, as the beams can be steered to remove old synapses and create new ones as they learn new information. Figure 4.1 shows a concept representation of the Spatial Phase Modulator (SPM) architecture. The interaction between two layers can be observed, where the top layer generates spikes that are received and processed by the lower layer.

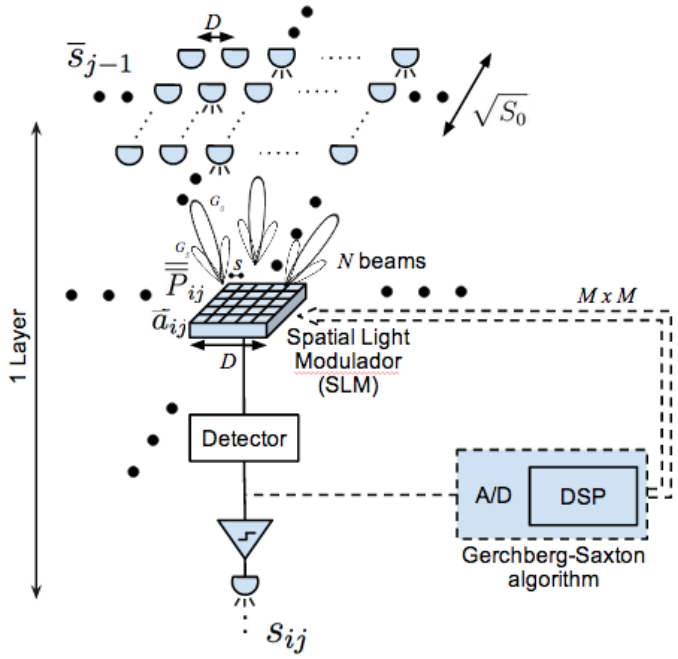
Figure 4.1: 3D representation of the SPM implementation.



4.1.1 Description

In this architecture, each model neuron is implemented by a SLM that processes the light received from the previous layer. The amount of light that crosses the SLM is measured with an optical detector. The output level of the detector is compared to a defined threshold, and if it is higher, the light-emitting diode (LED) at the lower end of the device emits a spike that may be captured by the next layer.

Figure 4.2: Diagram of the SPM implementation



Var	Description
D	Cognon spacing
S_0	# of synapses
N	# of excited inputs
s	SLM cell size
G_0	Main lobe gain
G_s	Side lobe gain
G	Ratio of strong synapse strength to weak synapse strength (G_0 / G_s)
$M \times M$	SLM array size
T_C	cognition time
T_L	learning time

A detailed diagram of the SPM architecture is shown in Figure 4.2. The top layer is identified as $\overline{s}_{j-1}$ following the previous chapter definition. It is represented by the set of output LEDs that light when the neuron that model fires. The LEDs are separated a distance D, which is also the width of each SLM.

Only one element of the bottom layer $\overline{s}_j$ is represented for clarity. At the top, the SLM generates a set of N beams, where each beam corresponds to a strengthened synapse. These beams implement the values of the projection matrix $\overline{\overline{P}}_{ij}$ and the strong synapses vector $\overline{a}_{ij}$ described in

the previous chapter. Also, the secondary lobes generate non-strengthened synapses. The difference in gain between the main lobe G_0 and the secondary lobes G_S is used to model the ratio of strong synapse strength to weak synapse strength $G = G_0/G_S$.

The amount of light that is received through the SLM is measured by the optical detector, implementing the weighted sum of the synapse inputs at the output of the detector. This output is then compared to the firing threshold H previously defined. If the output is higher than the threshold the neuron fires by lightening the LED at the bottom of the diagram.

4.1.1.1 Components

The presented SPM architecture uses 4 main elements that are described in this section.

SLM

Spatial light modulators (SLMs) are devices capable of converting data in electronic form (or sometimes in incoherent optical form) into spatially modulated coherent optical signals. These devices have been used since mid-1960s to modulate the intensity of laser beams for optical information processing applications. SLMs can be used in the input plane of optical processing systems to generate input images, holograms or interferograms; but also in the Fourier plane of analog optical processing systems [6]. Multiple technologies exist for these devices. Currently, the most important are liquid crystal SLMs.

Most screens of currently existing devices use liquid crystal displays. In such applications voltages applied to pixelated electrodes cause a change in the intensity of the light transmitted by or reflected from the display. Similar principles can be used to construct a spatial light modulator. Both displays and SLMs exploit the ability to change the transmittance of a liquid crystal by means of applied electric fields. Usually those fields are applied between the glass plates that contain the liquid crystal material using transparent conductive layers (indium tin oxide films) coated on the inside of the glass plates. In order to achieve alignment of the liquid crystal at the interface, the conductive layer is covered with a thin

alignment layer (often polyimide) which is subjected to polishing.

An example device that could be used for this system is Hamamatsu LCOS-SLM x10468, which is a pure phase SLM, based on Liquid Crystal on Silicon (LCOS) technology in which liquid crystal (LC) is controlled by a direct and accurate voltage, and can modulate a wavefront of light beam. This device has a resolution of 800 x 600 pixels and an efective area of 16 x 12 mm. The response time is 35 ms.

Detector

An optical detector, or photodetector, is a type of sensor that measures the intensity of light. In this system, the detector would measure the amount of light that arrives through the beams generated by the SLM. This would implement an equivalent to the weighted sum of the excitation pattern received from the previous layer neurons.

Signal Processor

During the learning phase, the system needs to change the phase values at the SLM. This process requires some calculations to estimate the desired phase distribution from the beam pattern that needs to be generated. Section 4.1.2 contains more details on the algorithm that should implement this part of the system.

LED

The last item of the device is an LED preceded by a comparator that checks if the signal measured by the detector is higher than the defined threshold. In that case the hardware neuron generates a spike by lighting the LED for a short interval, while the signals from the previous layer stay unchanged.

4.1.2 Learning

When a modeled neuron enters learning mode, for each new pattern that fires the neuron, the corresponding synapses need to be updated. This process is explained by the following algorithm:

(1) As the cognon is in learning phase, lower the firing threshold from the recognition level GH to H. This allows the neuron to fire for most of the inputs, and it learns each pattern that generates a firing event.

(2) If the cognon fires:

   (a) Find the previous layer lighted axons by scanning.

   (b) Strengthen the synapses that were excited by the pattern to G, by creating new beams pointing to the previous layer neurons.

   (c) Train the system to synthesize the old plus the new beams with the SLM, generating a new amplitude distribution by changing the phase distribution on the device.

This requires to change the phase distribution of the SLM to generate new beams in the directions where synapses should be strengthened. This problem is known as the phase problem, which consists of obtaining the right phase distribution for the SLM in order to generate a certain target amplitude pattern in the far-field.

Different algorithms have been proposed for solving this problem, which is known as phase retrieval. A complete review of different algorithms, including the most well-known Gerchberg-Saxton algorithm, can be found in [7]. In this section the most common Gerchberg-Saxton algorithm [10] is presented as a solution to be used in the SPM system.

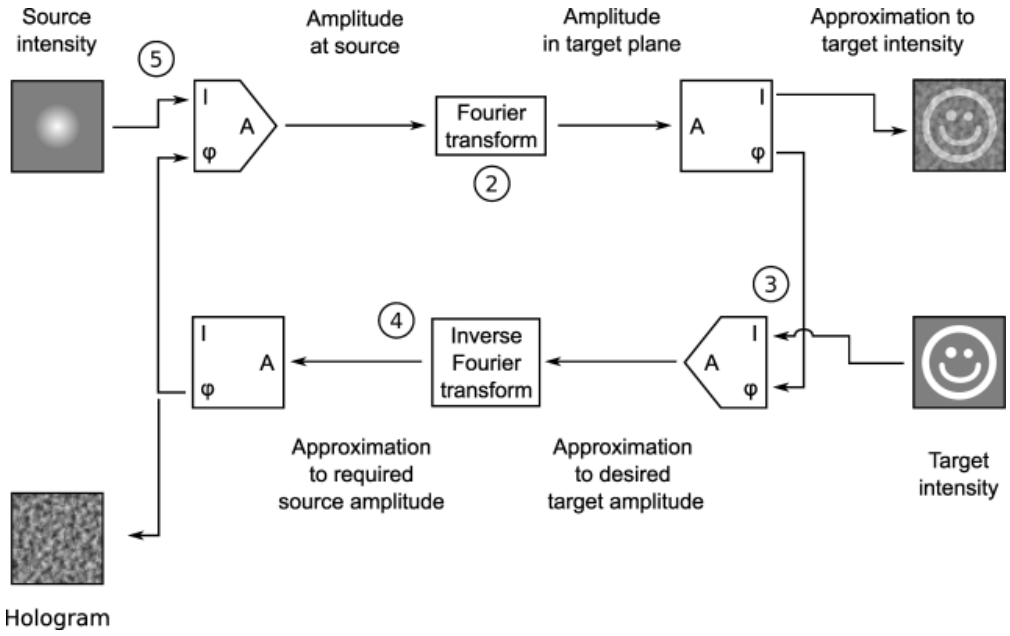
4.1.2.1 Gerchberg-Saxton algorithm

The Gerchberg-Saxton or error reduction algorithm solve the phase problem through the iteration of 4 steps, which can be summarized by the following iterative equation:

$$\Psi(x,y) = \lim_{n\to\infty} (\mathcal{P}_m \mathcal{F}^{-1} \mathcal{P}_F \mathcal{F})^n |\Psi(x,y) e^{i\phi(x,y)}| \tag{4.1}$$

where $\mathcal{F}$ denotes the Fourier transform, and $\mathcal{P}_F$ and $\mathcal{P}_m$, projection operators which adjust the amplitude so that the amplitude conditions in the Fourier and real space are respectively fulfilled. Figure 4.3 shows a diagram of this process.

Figure 4.3: Gerchberg-Saxton algorithm diagram



Under the given prerequisites (given real space amplitudes and evenly illuminated Fourier space), we start in the upper left corner. In the first iteration nothing is to adapt in real space so we can apply directly the Fourier transform to our pattern (if we start in the Fourier space, an initial guess of the phases would be necessary). In Fourier space the amplitudes of the transformed pattern will generally not match with the given ones. We set all amplitudes to the same value, but leave the phases unaltered. Next we transform the pattern back into real space. This completes the first iteration of the algorithm. But now the amplitudes have changed and we must adapt them to match the given pattern. This means we set the modulus of an illuminated pixel to one and of a dark pixel to zero. After that we use the Fourier transformation again and continue as before with amplitude adaption in Fourier space and so on. The algorithm is repeated as long as the convergence criterion is not met or a certain number of iterations is not reached.

The Gerchberg-Saxton algorithm is straight forward and easy to implement, but it has some flaws. It converges for example to the nearest minimum, which is not necessarily the global minimum of the deviation to the original image. In order to remedy those other issues, modified versions of Gerchberg-Saxton were introduced. An example of improved version of this algorithm is the relaxed averaged alternating reflections (RAAR) algorithm [17].

4.1.3 Performance

The performance of the system will be determined by the operation mode. During the recognition phase, the system does not need to change any parameters, and all the process is run by propagating light through the different layers of devices. This would allow high speed computations to be performed with the system during recognition phase, when not learning.

However, when the system is in learning phase, the speed would be more restricted. The calculations to update the phase of the SLM would need some time, probably in the order of microseconds, to run the Gerchberg-Saxton algorithm. We define this time as T_{GS}. After the desired phase distribution is calculated, the SLM would need to be updated, which is limited by the refresh period of the device, defined as T_{SLM}. Adding these two times we can obtain the

learning time of the system: $T_L = T_{GS} + T_{SLM}$.

4.2 Spatial Amplitude Modulator (SAM)

The other designed architecture uses commercially available optical devices, that include LEDs, lenses and LCDs, to recreate the details of the cognon model. Modeled neurons are arranged in 2D layers that are put one on top of the other. Each cognon is connected to 10,000 cognons of the previous layer using optical signals, which simulate the synapses through which the spikes are transmitted. Based on the initial estimations using current market technology, each cognon would have a size of 1 cm^2 and a layer spacing of 1 m would be needed. Figure 4.4 shows a 3D concept representation of the Spatial Amplitude Modulator (SAM) architecture. The interaction between two layers can be observed, where the top layer generates spikes that are received and processed by the lower layer.

4.2.1 Description

In this architecture, each synapse is modeled by a cell of an liquid crystal display (LCD), and a set of the cells simulate a neuron with all the synapses. Using a lens, the light from the previous layer is focused on the LCD and this applies the weighting to each of the input signals. Below the screen there is a diffuser that adds the light from all synapses and a detector the measures the total received light. As in the SPM architecture, the signal from the detector is compared to a threshold and the LED at the bottom is lighted, emulating a spike, if the signal is higher than the threshold.

A detailed diagram of the SAM architecture is shown in Figure 4.5. The top layer is identified as $\overline{s}_{j-1}$ following the previous chapter definition. It is represented by the set of output LEDs that light when the neuron that model fires. The LEDs are separated a distance D, which is also the width of each group of LCD cells.

The elements of layer $\overline{s}_j$ are represented at the bottom of the figure. Each element includes a lens that focuses the lights from the previous layer, an LCD region, a diffuser, a detector, a comparator and the final LED. Even though in the diagram the LCD regions are represented as

Figure 4.4: 3D representation of the SAM implementation.

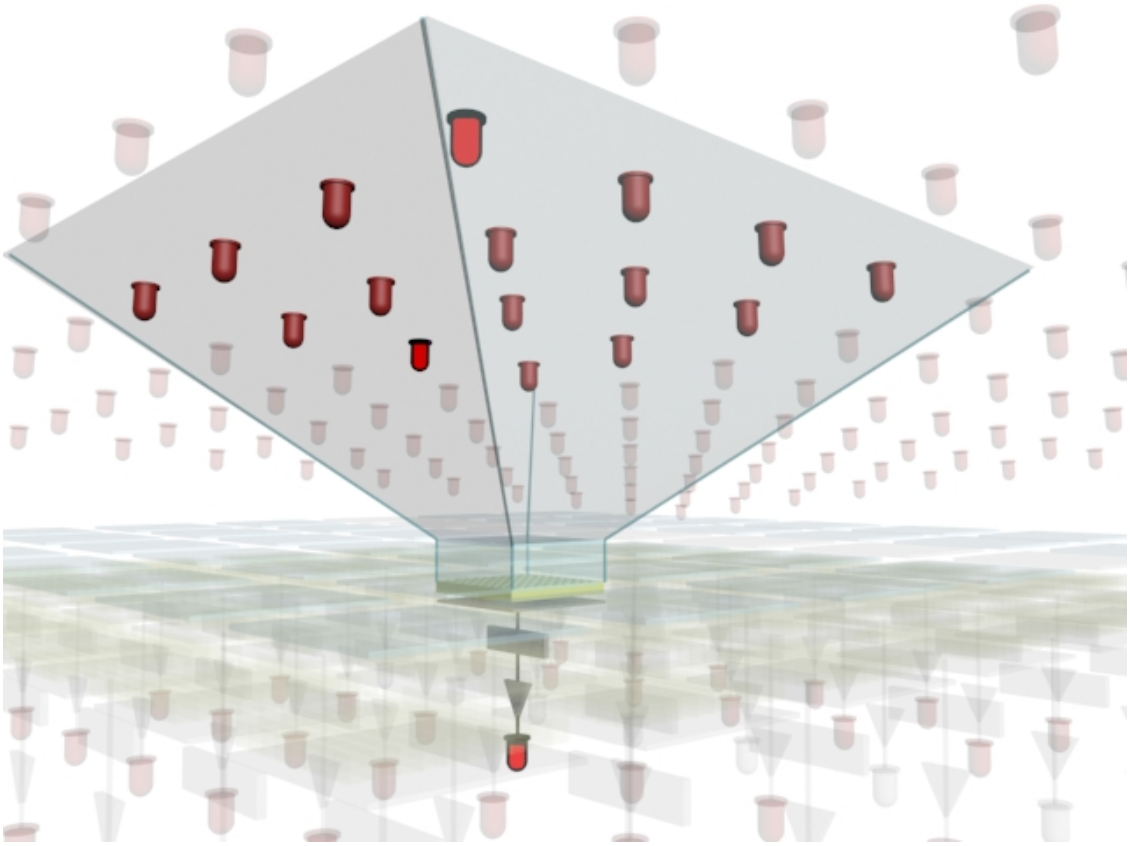
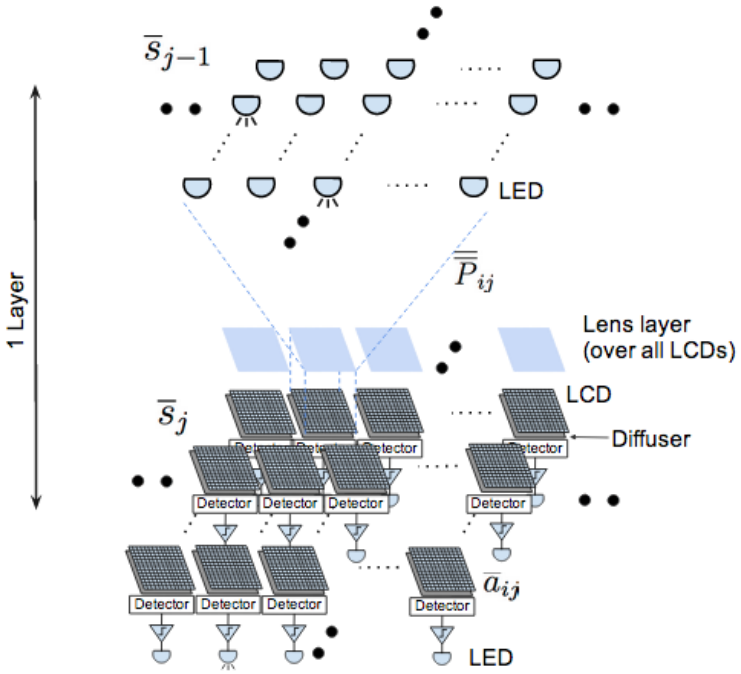


Figure 4.5: Diagram of the SAM implementation



separate LCDs, a practical implementation would share a high resolution screen between multiple neurons. The pixels of each region of the LCD would implement the projection matrix $\overline{\overline{P}}_{ij}$ and the strong synapses vector $\overline{a}_{ij}$ described in the previous chapter.

The amount of light that is received through the SLM is measured by the optical detector, implementing the weighted sum of the synapse inputs at the output of the detector. This output is then compared to the firing threshold H previously defined. If the output is higher than the threshold the neuron fires by lightening the LED at the bottom of the diagram.

4.2.1.1 Components

The presented SAM architecture uses 5 main elements that are described in this section.

Lens

The first item of the device is a plano-convex lens that focuses the light received from the previous layer on the LCD screen. The parameters that describe this item are shown in Table 4.1.

Table 4.1: Parameters and estimated values of the lens in the SAM architecture.

Parameter	Description	Value
D	Diameter	1 cm
f	Focal length	5 mm
$F\#$	f/D	0.5

LCD

The LCD could be obtained from any commercial available device with a high resolution screen. Current screens are capable of 300 dpi or better, what is equal to having about 0.1 mm per pixel. Together with the lens, this part is one of the size limiting components of the system. Table 4.2 shows the relevant parameters of this device.

Diffuser

Table 4.2: Parameters and estimated values of the LCD in the SAM architecture.

Parameter	Description	Value
A	Number of strong synapses	40
a	Pixel size	0.1 mm
G	Ratio strengthened pixel to non-strengthened	1.8

The diffuser adds the light that is received through the LCD in order to measure the sum of excited synapse, both strengthened and regular. Different technologies exist for these devices, with very different transmission efficiencies. The cheapest devices have very poor transmission rates, about 0.3, and would not be suitable for this application, as the LEDs required power would have to be increased. Table 4.3 shows the main parameter of the diffuser, with the estimated value for the best quality devices available.

Table 4.3: Parameters and estimated values of the diffuser in the SAM architecture.

Parameter	Description	Value
η_{diff}	Efficiency of transmission	0.9

Detector

An optical detector, or photodetector, is a type of sensor that measures the intensity of light. In this system, the detector would measure the amount of light that arrives through the pixel pattern in the LCD screen. This would implement an equivalent to the weighted sum of the excitation pattern received from the previous layer neurons. Table 4.4 shows the relevant parameters of this device.

LED

The last item of the device is an LED preceded by a comparator that checks if the signal measured by the detector is higher than the defined threshold. In that case the hardware neuron generates a spike by lighting the LED for a short interval, while the signals from

Table 4.4: Parameters and estimated values of the detector in the SAM architecture.

Parameter	Description	Value
P_R	Received power	
G_R	Gain pattern	
$\varnothing$	Diameter or active area	$3 - 100$ mm^2
NEP	Noise Equivalent Power	$8.6 \cdot 10^{-14}$ W/$\sqrt{Hz}$
R	Responsivity	0.62 A/W @ 950 nm

the previous layer stay unchanged. Table 4.5 shows the relevant parameters of the LED.

Table 4.5: Parameters and estimated values of the LED in the SAM architecture.

Parameter	Description	Value
P_T	Transmitted power	10 mW
G_T	Gain pattern	$FNBW \simeq 40°$
f_r	Firing rate	20 Hz
λ	Peak wavelength	900 nm
$\Delta\lambda$	Spectral width	50 nm

4.2.2 Learning

When a modeled neuron enters learning mode, for each new pattern that fires the neuron, the corresponding synapses need to be updated. In this architecture the process is easier, as there is no need to recalculate the phase distribution. This process is explained by the following algorithm:

(1) As the cognon is in learning phase, lower the firing threshold from the recognition level GH to H. This allows the neuron to fire for most of the inputs, and it learns each pattern that generates a firing event.

(2) If the cognon fires:

   (a) Find the previous layer lighted axons by scanning.

(b) Strengthen the synapses that were excited by the pattern to G, by adding them to the list of strengthened pixels.

(c) Update the LCD pixel values with the new strengthened synapses.

The process of finding the previous layer lighted axons can be optimized by using smart search algorithms. An option would be to use Welsch functions, by setting up a certain function on the LCD screen and measuring the received light power.

Another method that would be useful could be based on compressed sensing theory [5]. Given that the set of excited inputs in the excitation patterns is small compared to the total number of synapses, it can be considered as a sparse signal. Thus, this problem could be analyzed as a convex optimization problem. In order to obtain a solution with the minimum number of measurements, reducing the time to obtain the previous layer lighted inputs, would be to minimize the sum of excited inputs given that the weighted sum of the signals is equal to the measured power [3].

4.2.2.1 Inhibitory Synapses

In a network of cognon model neurons, it is obviously more efficient if only one or a few neighboring artificial neurons learn any given excitation pattern, rather than having excess duplication among many. Therefore it would be useful for an artificial neuron that recognizes one pattern to suppress its nearby neighbors' ability to learn the same pattern. One common hypothesis is that this is partly accomplished if some fraction of all axons inhibit firing of nearby neurons by using synapses having negative strength.

In order to implement these negative synapses in the SAM architecture, the system design needs to be modified, as when using incoherent light powers always add, and do not subtract. A method to obtain these desired negative strengths would be to use a two color system, where another subset of the LCD array would be used to implement negative synapses. The sum of negative synapses would be obtained using another detector per each cognon device. The result would have to be subtracted from the sum of the original detector before the comparator checks if

the result is higher than the threshold and whether the LED needs to be ligthed to fire the artificial neuron.

4.2.3 Performance

As in the SPM architecture, the performance of the system will be determined by the specifications of the components. In this case, the recognition speed can also be very high, as the computation is done through optical paths. However the refresh rate of the LCD screen might need to be considered in this operation mode.

Regarding the learning phase, the performance is limited by the refresh time of the screen T_{LCD} plus the time needed to find the excited inputs at each step of the learning process T_S. The total learning time for this device would be $T_L = T_{LCD} + T_S$.

Chapter 5

Conclusions

This work presents research on the development of intelligent computing systems using cortical spikes to communicate information between artificial neurons. The focus is on the cognon neural model, validating in software the existing results, and proposing two designs for hardware implementations of the model.

The cognon model, with the proposed extensions, is consistent with both observed neural behavior and sub-second learning and recall. It is an innovative approach that considers the information stored in the neuron as learnt patterns, and optimizes the parameters of the model based on the total recallable information. Software simulation shows promising results for this neural model. The original results are verified by a new implementation that prioritizes the clarity of the code and optimizes expensive operations.

No highly-interconnected, in the order of 10,000 synapses, scalable neural architecture implemented in hardware has been implemented yet, but multiple research groups are working on new systems. In this thesis, two new neural computing hardware architectures based on the cognon model are presented. One of the architectures is based on using SLMs as the main component to generate a set of beams that simulate synapses. The other hardware architecture uses an array of lens and commercially available LCD screens to create artificial synapses between layers of neurons. These architectures could lead to machines with intelligent capabilities, useful for processing high amounts of data in real-time and other applications that are difficult to solve with current computing technologies.

5.1 Suggestions of Future Research

Continuing on the results of this work, the Synapse Atrophy model, the second cognon extended model learning method, should also be considered and compared with the presented Synapse Strength model. The preliminary results of this model seem very promising. Later, this model could also be incorporated to the proposed hardware architectures. A logical extension to the existing results would be the simulation of a network of cognon model neurons, implementing in software a set of modelled neurons that interact propagating spikes.

Regarding the hardware architectures, the next step would consist of building a prototype hardware implementation of the designed presented in this thesis. Moreover, other architectures could be developed based on similar principles. Different communication systems could be used instead of optic signals. For example, radio-frequency strategies with coding schemes or frequency modulation.

Finally, building truly intelligent machines would require an understanding of how layers in the neocortex interact and form different regions associated with certain sensory inputs. This higher level architectures would have to be incorporated to the presented hardware architectures in order to achieve results similar to the capabilities of the mammalian brain.

Bibliography

[1] Rajagopal Ananthanarayanan, Steven K. Esser, Horst D. Simon, and Dharmendra S. Modha. The cat is out of the bag. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09, page 1, New York, New York, USA, 2009. ACM Press.

[2] Adam B Barrett and M C W van Rossum. Optimal learning rules for discrete synapses. PLoS computational biology, 4(11):1–7, November 2008.

[3] Stephen Boyd and Lieven Vandenberghe. Convex optimization, volume 25. Cambridge University Press, 2004.

[4] Santiago Ramón y Cajal. Comparative study of the sensory areas of the human cortex. 1899.

[5] D L Donoho. Compressed sensing. IEEE Transactions on Information Theory, 52(4):1289–1306, 2006.

[6] Uzi Efron. Spatial Light Modulator Technology: Materials, Devices, and Applications. CRC Press, 1995.

[7] J R Fienup. Phase retrieval algorithms: a comparison. Applied Optics, 21(15):2758–2769, 1982.

[8] SB Furber, Steve Temple, and AD Brown. High-performance computing for systems of spiking neurons. In AISB'06 workshop on GC5: Architecture of Brain and Mind, pages 29–36, 2006.

[9] Steve Furber and Steve Temple. Neural systems engineering. Journal of the Royal Society, Interface / the Royal Society, 4(13):193–206, April 2007.

[10] RW Gerchberg. A practical algorithm for the determination of phase from image and diffraction plane pictures. Optik, 35(2):237–246, 1972.

[11] Bruce Graham and David Willshaw. Probabilistic Synaptic Transmission in the Associative Net. Neural Computation, 11(1):117–137, January 1999.

[12] Jeff Hawkins and Sandra Blakeslee. On intelligence. Times Books, New York, 2004.

[13] E M Izhikevich. Simple model of spiking neurons. IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council, 14(6):1569–72, January 2003.

[14] Eugene M Izhikevich. Which model to use for cortical spiking neurons? IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council, 15(5):1063–70, September 2004.

[15] Eric Kandel, James Schwartz, and Thomas Jessell. Principles of Neural Science. McGraw-Hill Medical, 2000.

[16] Christof Koch. Biophysics of Computation: Information Processing in Single Neurons. Oxford University Press, USA, 2004.

[17] D Russell Luke. Relaxed Averaged Alternating Reflections for Diffraction Imaging. Inverse Problems, 21(1):13, 2004.

[18] Henry Markram. The blue brain project. Nature reviews. Neuroscience, 7(2):153–60, February 2006.

[19] Henry Markram, Joachim Lübke, Michael Frotscher, and Bert Sakmann. Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs. Science, 275(5297):213–215, January 1997.

[20] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics, 5(4):115–133, December 1943.

[21] Paul Merolla, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and Dharmendra S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In 2011 IEEE Custom Integrated Circuits Conference (CICC), pages 1–4. IEEE, September 2011.

[22] B Pakkenberg and H J Gundersen. Neocortical neuron number in humans: effect of sex and age. The Journal of comparative neurology, 384(2):312–20, July 1997.

[23] Gerhard Roth and Ursula Dicke. Evolution of the brain and intelligence. Trends in cognitive sciences, 9(5):250–7, May 2005.

[24] David H. Staelin and Carl H. Staelin. Models for Neural Spike Computation and Cognition. 2011.

[25] Rufin VanRullen, Rudy Guyonneau, and Simon J Thorpe. Spike times make sense. Trends in neurosciences, 28(1):1–4, January 2005.

Appendix A

Source code

A.1 cognon_basic.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

import numpy as np


class Word(object):
    """A Word contains a list of those input synapses that fired for the most
    recent given excitation pattern.

    Attributes:
        offset: A set containing the syanpses that fired.
    """

    def __init__(self, fired_syn=[]):
```

```
        """Inits Word class.

        Args:
            fired_syn: List of input synapses that fired. Can only contain
            positive values.
        """
        if len(fired_syn) > 0 and sorted(fired_syn)[0] < 0:
            raise ValueError('offset values have to be positive')
        self.offset = set(fired_syn)



class Neuron(object):
    """Models a CB neuron.

    Attributes:
        SO: Number of synapses.
        H: Number of synapses needed to fire a neuron.
        G: Ratio of strong synapse strength to weak synapse strength, binary
            approximation.
        training: whether the neuron is in training mode.
    """

    def __init__(self, SO = 16, H = 4.0, G = 2.0):
        """Inits Neuron class.

        Args:
            SO: Number of synapses.
            H: Number of synapses needed to fire a neuron.
            G: Ratio of strong synapse strength to weak synapse strength,
                binary approximation.
        """
        self.SO = SO
        self.H = H
        self.G = G
        self.strength = np.ones(SO)
        self.training = False


    def expose(self, w):
        """Models how the neuron reacts to excitation patterns, and how it computes
       whether or not to fire.

       Expose computes the weighted sum of the input word, and the neuron fires if
       that sum meets or exceeds a threshold. The weighted sum is the sum of the
       SO element-by-element products f the most recent neron vector, the current
       word, and the neuron frozen Boolean vector.
```

```
    Args:
        w: A Word to present to the neuron.

    Returns:
        A Boolean indicating whether the neuron will fire or not.
    """
     # Compute the weighted sum of the firing inputs
     s = self.strength[list(w.offset)].sum()
     if self.training:
         return s >= self.H
     else:
         return s >= self.H*self.G


def train(self, w):
    """Trains a neuron with an input word.

    To train a neuron, "train" is called for each word to be recognized. If the
    neuron fires for that word then all synapses that contributed to that
    firing have their strengths irreversibly increased to G.

    Args:
        w: A Word to train the neuron with.

    Returns:
        A Boolean indicating whether the neuron fired or not.
    """
     if not self.training:
         print "[WARN] train(w) was called when not in training mode."
         return False

     if not self.expose(w): return False

     # Set the srength for participating synapses to G
     self.strength[list(w.offset)] = self.G

     return True


def start_training(self):
    """Set the neuron in training mode.
    """
     self.training = True


def end_training(self):
```

```
        """Set the neuron in recognition mode.

        Once the training is complete, the neuron's threshold value H is set
        to H*G.
        """
        self.training = False
```

A.2 test_cognon_basic.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

from cognon_basic import Neuron
from cognon_basic import Word

from nose.tools import assert_false
from nose.tools import assert_in
from nose.tools import assert_true
from nose.tools import eq_
from nose.tools import raises


class TestWord:

    def test_empty(self):
        w = Word()
        eq_(len(w.offset), 0)

    @raises(ValueError)
    def test_negative_offset(self):
        w = Word([-1])
```

```
    def test_fire_1_3_8(self):
        w = Word([1,3,8])
        eq_(len(w.offset), 3)
        assert_in(1, w.offset)
        assert_in(3, w.offset)
        assert_in(8, w.offset)


class TestNeuron:

    def test_defaults(self):
        n = Neuron()
        eq_(n.S0, 16)
        eq_(n.H, 4.0)
        eq_(n.G, 2.0)
        eq_(len(n.strength), n.S0)
        assert_false(n.training)

    def test_expose_not_training(self):
        n = Neuron(S0 = 16, H = 4.0, G = 2.0)

        w1 = Word([1,6,9])
        assert_false(n.expose(w1))

        w2 = Word([1,3,4,5,6,8,9,14])
        assert_true(n.expose(w2))

    @raises(IndexError)
    def test_expose_index_error(self):
        n = Neuron(S0 = 16)
        w = Word([16])
        n.expose(w)

    def test_train(self):
        n = Neuron(16, 4.0, 2.0)
        wA = Word([1,6,9,14])
        wB = Word([3,4,9,13])

        n.start_training()
        assert_true(n.train(wA))
        assert_true(n.train(wB))
        n.end_training()

        wD = Word([2,6,12,14])
        wE = Word([3,7,9,13])
        assert_false(n.expose(wD))
```

```
        assert_false(n.expose(wE))

        wF = Word([1,4,9,14])
        assert_true(n.expose(wF))

    def test_train_not_training(self):
        n = Neuron()
        w = Word()
        assert_false(n.train(w))
```

A.3 cognon_extended.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

from collections import namedtuple
import numpy as np
import random


class Synapse(namedtuple('Synapse', ['offset', 'delay'])):
    """A Synapse represents a connection between the neuron's input dendrites
    and the output axons of other neurons.

    Attributes:
        offset: Identifies a synapse of the neuron.
        delay: Represents the time the signal takes to traverse the axon to
        reach the synapse. Takes a value in range(D1).
    """
    pass
```

```
class Word(object):
    """An input Word represents the input signals to the neuron for a time
    period.

    A Word contains a list of those input synapses that fired for the
    most recent given excitation pattern.

    Attributes:
        synapses: A set of pairs containing the syanpses that fired and the
            associated delay.
    """

    def __init__(self, fired_syn=[]):
        """Inits Word class.

        Args:
            fired_syn: List of pairs of input synapses that fired and
                associated delays. Can only contain positive synapse offset
                values.
        """
        if len(fired_syn) > 0 and sorted(fired_syn)[0][0] < 0:
            raise ValueError('synapse offset values have to be positive')
        self.synapses = [Synapse(*s) for s in fired_syn]



class WordSet(object):
    """An array of Words.

    Wordset is simply an array of Word instances, which may also store
    information regarding the delay slot learned for the word during training.

    Attributes:
        words: Array of Word instances.
        delays: Delay slots learned for each word during training.
    """

    def __init__(self, num_words, word_length, num_delays, num_active=None,
                       refractory_period=None):
        """Inits WordSet class.

       Args:
           num_words: Number of Words to initialize the WordSet with.
           word_length: Number of synapses in a Word.
           num_delays: Number of delay slots.
```

```
            num_active: Number of active synapses per word.
            refractory_period: Average number of different patterns presented
                               before a given neuron fires.
        """
        # Distribution of the number of active synapses per word?
        if not refractory_period:
            # fixed: N
            N_array = np.empty(num_words, int)
            N_array.fill(num_active)
        else:
            # binomial: B(S0, 1/R)
            N_array = np.random.binomial(word_length, 1.0/refractory_period,
                                         num_words)

        # Generate the set of words and set delays to 0
        synapses = range(word_length)
        self.words = [Word(zip(
                random.sample(synapses, N),              # active synapses
                np.random.randint(num_delays, size=N)))  # active delays
            for N in N_array]
        self.delays = [0] * num_words



class Neuron(object):
    """Models a CE neuron.

    Attributes:
        S0: Number of synapses.
        H: Number of synapses needed to fire a neuron.
        G: Ratio of strong synapse strength to weak synapse strength, binary
            approximation.
        C: Number of dendrite compartments capable of firing independently.
        D1: Number of possible time slots where neurons can produce spikes.
        D2: Number of different time delays available between two neural
            layers.
        synapses: Represents a connection between the neuron's input dendrites
            and the output axons of other neurons. Each row of the array
            contains 3 fields:
            - strength: Strength of the synapse.
            - delay: Represents the time the signal takes to traverse the axon
              to reach the synapse. Takes a value in range(D2).
            - container: The dendrite compartment of this synapse.
        training: whether the neuron is in training mode.
    """

    def __init__(self, S0 = 200, H = 5.0, G = 2.0, C = 1, D1 = 4, D2 = 7):
```

```
        """Inits Neuron class.

        Args:
            S0: Number of synapses.
            H: Number of synapses needed to fire a neuron.
            G: Ratio of strong synapse strength to weak synapse strength,
                binary approximation.
            C: Number of dendrite compartments capable of firing independently.
            D1: Number of possible time slots where neurons can produce spikes.
            D2: Number of different time delays available between two neural
                layers.
        """
        self.S0 = S0
        self.H = H
        self.G = G
        self.C = C
        self.D1 = D1
        self.D2 = D2
        self.training = False
        self.synapses = np.zeros(S0, dtype='float32,uint16,uint16')
        self.synapses.dtype.names = ('strength', 'delay', 'container')
        self.synapses['strength'] = 1.0
        self.synapses['delay'] = np.random.randint(D2, size=S0)
        self.synapses['container'] = np.random.randint(C, size=S0)


    def expose(self, w):
        """Models how the neuron reacts to excitation patterns, and how it
        computes whether or not to fire.

        Expose computes the weighted sum of the input word, and the neuron fires
        if that sum meets or exceeds a threshold. The weighted sum is the sum of
        the S0 element-by-element products of the most recent neuron vector and
        the current word.

        Args:
            w: A Word to present to the neuron.

        Returns:
            A 3-element tuple containing:
                0. A Boolean indicating whether the neuron fired or not.
                1. The delay in which the neuron has fired.
                2. The container where the firing occurred.
        """
        offsets = [syn.offset for syn in w.synapses]
        delays  = [syn.delay  for syn in w.synapses]
        synapses = self.synapses[offsets]
```

```
        # Iterate over delays until neuron fires
        for d in range(self.D1 + self.D2):
            delay_indices = (synapses['delay'] + delays) == d

            # Compute the weighted sum of the firing inputs for each container
            for c in range(self.C):
                container_indices = synapses['container'] == c
                indices = delay_indices & container_indices
                s = synapses['strength'][indices].sum()

                # Check if the container has fired
                if (self.training and s >= self.H) or s >= self.H*self.G:
                    return (True, d, c)

        # If no container has fired for any delay
        return (False, None, None)


    def train(self, w):
        """Trains a neuron with an input word.

        To train a neuron, "train" is called for each word to be recognized. If
        the neuron fires for that word then all synapses that contributed to
        that firing have their strengths irreversibly increased to G.

        Args:
            w: A Word to train the neuron with.

        Returns:
            A Boolean indicating whether the neuron fired or not.
        """
        if not self.training:
            print "[WARN] train(w) was called when not in training mode."
            return False

        fired, delay, container = self.expose(w)
        if not fired: return False

        # Update the synapses that contributed to the firing
        offsets = [s.offset for s in w.synapses]
        delays  = [syn.delay  for syn in w.synapses]

        synapses = self.synapses[offsets]

        delay_indices = (synapses['delay'] + delays) == delay
        container_indices = synapses['container'] == container
```

```
        active_indices = delay_indices & container_indices

        indices = np.zeros(self.S0, dtype=bool)
        indices[offsets] = active_indices

        self.synapses['strength'][indices] = self.G

        return True


    def start_training(self):
        """Set the neuron in training mode.
       """
        self.training = True


    def finish_training(self):
        """Set the neuron in recognition mode.

       Once the training is complete, the neuron's threshold value H is set
       to H*G.
       """
        self.training = False
```

A.4 test_cognon_extended.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

from cognon_extended import Neuron
from cognon_extended import Synapse
```

```
from cognon_extended import Word
from cognon_extended import WordSet

from nose.tools import assert_false
from nose.tools import assert_greater_equal
from nose.tools import assert_in
from nose.tools import assert_is_none
from nose.tools import assert_less
from nose.tools import assert_less_equal
from nose.tools import assert_true
from nose.tools import eq_
from nose.tools import ok_
from nose.tools import raises


class TestSynapse:

    @raises(TypeError)
    def test_construct_requires_args(self):
        s = Synapse()

    def test_named_attributes(self):
        s = Synapse(1, 0)
        eq_(s.offset, 1)
        eq_(s.delay, 0)


class TestWord:

    def test_empty(self):
        w = Word()
        eq_(len(w.synapses), 0)

    @raises(ValueError)
    def test_negative_synapse_offset(self):
        w = Word([(-1, 0)])

    def test_fire_1_3_8(self):
        w = Word([(1,0),(3,0),(8,0)])
        eq_(len(w.synapses), 3)
        assert_in((1,0), w.synapses)
        assert_in((3,0), w.synapses)
        assert_in((8,0), w.synapses)

    def test_delay_0(self):
        w = Word([(1,0),(3,0),(8,0)])
        for offset, delay in w.synapses:
```

```
            eq_(delay, 0)


class TestWordSet:

    def test_small(self):
        num_words = 5
        word_length = 16
        num_delays = 4
        num_active = 4

        ws = WordSet(num_words, word_length, num_delays, num_active)

        eq_(len(ws.words), num_words)
        eq_(len(ws.delays), num_words)

        for word in ws.words:
            eq_(len(word.synapses), num_active)
            for synapse in word.synapses:
                assert_greater_equal(synapse.offset, 0)
                assert_less(synapse.offset, word_length)
                assert_greater_equal(synapse.delay, 0)
                assert_less(synapse.delay, num_delays)

    def test_refractory_period(self):
        num_words = 5
        word_length = 16
        num_delays = 4
        num_active = None
        refractory_period = 4

        ws = WordSet(num_words, word_length, num_delays, num_active,
                refractory_period)

        eq_(len(ws.words), num_words)
        eq_(len(ws.delays), num_words)

        for word in ws.words:
            for synapse in word.synapses:
                assert_greater_equal(synapse.offset, 0)
                assert_less(synapse.offset, word_length)
                assert_greater_equal(synapse.delay, 0)
                assert_less(synapse.delay, num_delays)


class TestNeuron:
```

```
def test_defaults(self):
    n = Neuron()
    eq_(n.S0, 200)
    eq_(n.H, 5.0)
    eq_(n.G, 2.0)
    eq_(n.C, 1)
    eq_(n.D1, 4)
    eq_(n.D2, 7)
    assert_false(n.training)
    eq_(len(n.synapses), n.S0)
    assert_true((n.synapses['strength'] == 1.0).all())
    assert_true((n.synapses['delay'] >= 0).all())
    assert_true((n.synapses['delay'] < n.D2).all())
    assert_true((n.synapses['container'] >= 0).all())
    assert_true((n.synapses['container'] < n.C).all())

def test_attributes_in_range(self):
    n = Neuron()
    assert_greater_equal(n.H, 1.0)
    assert_greater_equal(n.C, 1)
    assert_less_equal(n.D1, n.D2)
    assert_true((n.synapses['strength'] >= 0.0).all())

def test_expose_not_training(self):
    n = Neuron(S0 = 16, H = 4.0, G = 2.0, C = 1, D1 = 1, D2 = 1)

    w = Word([(1,0), (6,0), (9,0)])
    fired, delay, container = n.expose(w)
    assert_false(fired)
    assert_is_none(delay)
    assert_is_none(container)

    w = Word([(1,0), (3,0), (4,0), (5,0), (6,0), (8,0), (9,0), (14,0)])
    fired, delay, container = n.expose(w)
    assert_true(fired)
    eq_(delay, 0)
    eq_(container, 0)

@raises(IndexError)
def test_expose_index_error(self):
    n = Neuron(S0 = 16)
    w = Word([(16,0)])
    n.expose(w)

def test_expose_multiple_containers(self):
    n = Neuron(S0 = 16, H = 2.0, G = 2.0, C = 3, D1 = 1, D2 = 1)
```

```
        # Set container assignment manually to remove randomness
        n.synapses['container'][ 0:10] = 0
        n.synapses['container'][10:14] = 1
        n.synapses['container'][14:16] = 2

        w = Word([(1,0), (2,0), (6,0)])
        fired, delay, container = n.expose(w)
        assert_false(fired)
        assert_is_none(delay)
        assert_is_none(container)

        w = Word([(1,0), (2,0), (3,0), (4,0), (5,0), (6,0)])
        fired, delay, container = n.expose(w)
        assert_true(fired)
        eq_(delay, 0)
        eq_(container, 0)

        w = Word([(10,0), (11,0), (12,0), (13,0)])
        fired, delay, container = n.expose(w)
        assert_true(fired)
        eq_(delay, 0)
        eq_(container, 1)

        w = Word([(14,0), (15,0)])
        fired, delay, container = n.expose(w)
        assert_false(fired)
        assert_is_none(delay)
        assert_is_none(container)

    def test_expose_with_delays(self):
        n = Neuron(S0 = 16, H = 2.0, G = 2.0, C = 1, D1 = 2, D2 = 3)

        # Set delay assignment manually to remove randomness
        n.synapses['delay'][ 0:10] = 0
        n.synapses['delay'][10:14] = 1
        n.synapses['delay'][14:16] = 2

        w = Word([(1,0), (2,0), (6,0)])
        fired, delay, container = n.expose(w)
        assert_false(fired)
        assert_is_none(delay)
        assert_is_none(container)

        w = Word([(1,0), (2,0), (3,0), (4,0), (5,0), (6,0)])
        fired, delay, container = n.expose(w)
        assert_true(fired)
        eq_(delay, 0)
```

```
        eq_(container, 0)

        w = Word([(1,1), (2,1), (3,1), (4,1), (5,0), (6,0)])
        fired, delay, container = n.expose(w)
        assert_true(fired)
        eq_(delay, 1)
        eq_(container, 0)

        w = Word([(1,0), (2,0), (3,0), (4,1), (5,1), (6,1)])
        fired, delay, container = n.expose(w)
        assert_false(fired)
        assert_is_none(delay)
        assert_is_none(container)

        w = Word([(10,1), (11,1), (12,1), (13,1)])
        fired, delay, container = n.expose(w)
        assert_true(fired)
        eq_(delay, 2)
        eq_(container, 0)

        w = Word([(12,0), (13,0), (14,0), (15,0)])
        fired, delay, container = n.expose(w)
        assert_false(fired)
        assert_is_none(delay)
        assert_is_none(container)

    def test_train(self):
        n = Neuron(SO = 16, H = 4.0, G = 2.0, C = 1, D1 = 1, D2 = 1)

        # Train neuron with 2 patterns
        wA = Word([(1,0), (6,0), (9,0), (14,0)])
        wB = Word([(3,0), (4,0), (9,0), (13,0)])

        n.start_training()
        assert_true(n.train(wA))
        assert_true(n.train(wB))
        n.finish_training()

        # Test recognition
        wD = Word([(2,0), (6,0), (12,0), (14,0)])
        fired, delay, container = n.expose(wD)
        assert_false(fired)

        wE = Word([(3,0), (7,0), (9,0), (13,0)])
        fired, delay, container = n.expose(wE)
        assert_false(fired)
```

```
        wF = Word([(1,0), (4,0), (9,0), (14,0)])
        fired, delay, container = n.expose(wF)
        assert_true(fired) # False alarm

    def test_train_not_training(self):
        n = Neuron()
        w = Word()
        assert_false(n.train(w))

    def test_train_with_delays(self):
        n = Neuron(SO = 16, H = 4.0, G = 2.0, C = 1, D1 = 2, D2 = 2)

        # Fix neuron delays manually for the test
        n.synapses['delay'] = 1
        n.synapses['delay'][1] = 0
        n.synapses['delay'][14] = 0

        # Train neuron with 2 patterns
        wA = Word([(1,1), (6,0), (9,0), (14,1)])
        wB = Word([(3,0), (4,0), (9,0), (13,0)])

        n.start_training()
        assert_true(n.train(wA))
        assert_true(n.train(wB))
        n.finish_training()

        # Recognize
        wD = Word([(2,0), (6,0), (12,0), (14,0)])
        fired, delay, container = n.expose(wD)
        assert_false(fired)

        wE = Word([(1,1), (3,0), (9,0), (13,0)])
        fired, delay, container = n.expose(wE)
        assert_true(fired) # False alarm

        wF = Word([(1,0), (4,1), (7,0), (9,0), (11,0), (14,0)])
        fired, delay, container = n.expose(wF)
        assert_false(fired)
```

A.5 run_experiment.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

from cognon_extended import Neuron
from cognon_extended import WordSet

from math import log
from multiprocessing import Pool
import numpy as np
import os


class Alice(object):

    def train(self, neuron, wordset):
        neuron.start_training()

        for word in wordset.words:
            fired = neuron.train(word)

        neuron.finish_training()



class Bob(object):

    def __init__(self):
        self.true_true = 0   # trained and fired (ok)
        self.true_false = 0  # trained but not fired (false negative)
        self.false_true = 0  # not trained but fired (false positive)
        self.false_false = 0 # not trained and not fired (ok)

    def test(self, neuron, train_wordset, test_wordset):
        # Check the training set
        for word in train_wordset.words:
            fired, delay, container = neuron.expose(word)
            if fired:
```

```
                self.true_true += 1
            else:
                self.true_false += 1

        # Check the test set
        #num_active = len(train_wordset.words[0].synapses[0])
        #test_wordset = Wordset(num_test_words, neuron.S0, neuron.D1, num_active)
        for word in test_wordset.words:
            fired, delay, container = neuron.expose(word)
            if fired:
                self.false_true += 1
            else:
                self.false_false += 1



class Configuration(object):

    def __init__(self):
        self.neuron_params()
        self.test_params()


    def neuron_params(self, C = 1, D1 = 4, D2 = 7, Q = 40, G = 2, H = 5):
        self.H = H   # Num. of synapses needed to fire a neuron
        self.G = G   # Ratio of strong synapse strength to weak synapse s.
        self.C = C   # Num. of dendrite compartments
        self.D1 = D1 # Num. of posible time slots where spikes can happen
        self.D2 = D2 # Num. of time delays available between two layers
        self.Q = Q   # Q = S0/(H*R*C)


    def test_params(self, num_active = 4, R = None, w = 100, num_test_words = 0):
        self.num_active = num_active # Num. of active synapses per word
        self.R = R   # Avg. num. of patterns per afferent synapse spike
        self.w = w   # Num. of words to train the neuron with
        self.num_test_words = num_test_words # Num. of words to test


    @property
    def S0(self):
        if self.R:
            return int(self.Q * self.H * self.C * self.R)
        else:
            return int(self.Q * self.H * self.C)
```

```
class Cognon(object):

    def __call__(self, config):

        # If the OS is Unix, reseed the random number generator
        # http://stackoverflow.com/a/6914470/2643281
        if os.name == "posix":
            np.random.seed()

        return self.run_experiment(config)


    def run_configuration(self, config, repetitions):

        # Ensure that at least 10,000 words are learnt
        MIN_LEARN_WORDS = 10000
        #MIN_LEARN_WORDS = 1
        if repetitions * config.w < MIN_LEARN_WORDS:
            N = MIN_LEARN_WORDS/config.w
        else:
            N = repetitions

        # Ensure that at least 1,000,000 words are tested
        MIN_TEST_WORDS = 1000000
        if not config.num_test_words:
            config.num_test_words = MIN_TEST_WORDS/N

        # Run all the experiments
        #values = [self.run_experiment(config) for i in xrange(N)]
        pool = Pool(processes=20)
        values = pool.map(Cognon(), [config,]*N)

        # Store the results in a NumPy structured array
        names = ('pL', 'pF', 'L')
        types = [np.float64,] * len(values)
        r = np.array(values, dtype = zip(names, types))

        return r


    def run_experiment(self, cfg):

        # create a neuron instance with the provided parameters
        neuron = Neuron(cfg.S0, cfg.H, cfg.G, cfg.C, cfg.D1, cfg.D2)

        # create the training and test wordsets
```

```
        train_wordset = WordSet(cfg.w, cfg.S0, cfg.D1, cfg.num_active, cfg.R)
        test_wordset = WordSet(cfg.num_test_words, cfg.S0, cfg.D1,
                               cfg.num_active, cfg.R)

        # create Alice instance to train the neuron
        alice = Alice()
        alice.train(neuron, train_wordset)

        # create a Bob instance to test the neuron
        bob = Bob()
        bob.test(neuron, train_wordset, test_wordset)

        # results
        pL = bob.true_true/float(cfg.w)
        pF = bob.false_true/float(cfg.num_test_words)

        # L  = w*((1-pL)*log2((1-pL)/(1-pF)) + pL*log2(pL/pF)) bits
        L = 0
        if pL == 1.0:
            if pF != 0:
                L = -cfg.w*log(pF)/log(2.0)
            else:
                L = cfg.w
        elif pL > pF and pF != 0:
            L = cfg.w/log(2.0) * \
                (log(1.0 - pL) - log(1.0 - pF) +
                 pL * (log(1.0 - pF) - log(1.0 - pL) + log(pL) - log(pF)))

        return pL, pF, L
```

A.6 test_run_experiment.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
# limitations under the License.
#

from run_experiment import Alice
from run_experiment import Bob
from run_experiment import Configuration
from run_experiment import Cognon

from cognon_extended import Neuron
from cognon_extended import Word
from cognon_extended import WordSet

from nose.tools import assert_false
from nose.tools import assert_greater_equal
from nose.tools import assert_in
from nose.tools import assert_is_none
from nose.tools import assert_less
from nose.tools import assert_less_equal
from nose.tools import assert_true
from nose.tools import eq_
from nose.tools import ok_
from nose.tools import raises

from unittest.case import SkipTest


class TestAlice:

    def test_train(self):
        n = Neuron(S0 = 16, H = 4.0, G = 2.0, C = 1, D1 = 1, D2 = 1)

        wA = Word([(1,0), (6,0), (9,0), (14,0)])
        wB = Word([(3,0), (4,0), (9,0), (13,0)])
        wordset = WordSet(num_words = 2, word_length = 16, num_delays = 1,
                          num_active = 4)
        wordset.words = [wA, wB]

        alice = Alice()
        alice.train(n, wordset)

        # Test recognition
        wD = Word([(2,0), (6,0), (12,0), (14,0)])
        fired, delay, container = n.expose(wD)
        assert_false(fired)

        wE = Word([(3,0), (7,0), (9,0), (13,0)])
```

```
        fired, delay, container = n.expose(wE)
        assert_false(fired)

        wF = Word([(1,0), (4,0), (9,0), (14,0)])
        fired, delay, container = n.expose(wF)
        assert_true(fired) # False alarm


class TestBob:

    def test_test(self):
        n = Neuron(S0 = 16, H = 4.0, G = 2.0, C = 1, D1 = 1, D2 = 1)

        wA = Word([(1,0), (6,0), (9,0), (14,0)])
        wB = Word([(3,0), (4,0), (9,0), (13,0)])
        train_wordset = WordSet(num_words = 2, word_length = 16,
                                num_delays = 1, num_active = 4)
        train_wordset.words = [wA, wB]

        alice = Alice()
        alice.train(n, train_wordset)

        wD = Word([(2,0), (6,0), (12,0), (14,0)])
        wE = Word([(3,0), (7,0), (9,0), (13,0)])
        wF = Word([(1,0), (4,0), (9,0), (14,0)]) # False alarm
        test_wordset = WordSet(num_words = 3, word_length = 16,
                               num_delays = 1, num_active = 4)
        test_wordset.words = [wD, wE, wF]

        bob = Bob()
        bob.test(n, train_wordset, test_wordset)

        eq_(bob.true_true, 2)
        eq_(bob.true_false, 0)
        eq_(bob.false_true, 1)
        eq_(bob.false_false, 2)


class TestConfiguration:

    def test_config(self):
        raise SkipTest


class TestCognon:

    def test_cognon(self):
```

```
        raise SkipTest
```

A.7 create_tables.py

```
# Copyright 2013 Pau Haro Negre
# based on C++ code by Carl Staelin Copyright 2009-2011
#
# See the NOTICE file distributed with this work for additional information
# regarding copyright ownership.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

from run_experiment import Cognon
from run_experiment import Configuration


def run_table_row(w, active, C, D1, D2, Q, R, G, H):
    repetitions = 20

    config = Configuration()
    config.neuron_params(C, D1, D2, Q, G, H)
    #config.test_params(active, R, w, 5000)
    config.test_params(active, R, w)

    cognon = Cognon()
    return cognon.run_configuration(config, repetitions)


def table21():
    print "%%%%%%%%%%%%%%%"
    print "% Table 2.1. %"
    print "%%%%%%%%%%%%%%%"
    print
              # N | H |  S0  |  w  | G #
    table21_row( 4,  4,    10,   1, 100)
    table21_row( 5,  4,    10,   1, 100)
```

```
    table21_row( 4,  4,    10,   2, 100)
    table21_row(10, 10,   100,   4, 100)
    table21_row(11, 10,   100,   4, 100)
    table21_row(11, 10,   100,   5, 100)
    table21_row(11, 10,  1000,  60, 100)
    table21_row(11, 10, 10000, 600, 100)
    table21_row(22, 20, 10000, 450, 100)

    print "\t\\midrule"

    table21_row(10, 10,   100,   6, 1.5)
    table21_row(11, 10,  1000,  15, 1.5)
    table21_row(11, 10, 10000, 160, 1.5)
    table21_row(14, 10, 10000,  10, 1.5)

def table21_row(N, H, S, w, G):
    r = run_table_row(w, N, 1, 1, 1, S/float(H), None, G, float(H))
    pF_mean = r['pF'].mean()*100
    pF_std  = r['pF'].std()*100
    L_mean  = r['L'].mean()
    L_S0    = L_mean/S
    txt = ("\t{:.2f} & {} & {} & {:,} & {} & {} & {:.1f} & {:.2f} \\\\ "
           "% {:.3f}")
    print txt.format(pF_mean, N, H, S, w, G, L_mean, L_S0, pF_std)


def table23():
    print "%%%%%%%%%%%%%%%"
    print "% Table 2.3. %"
    print "%%%%%%%%%%%%%%%"
    print
              #  H |  G |  S0  |  R  | w #
    table23_row( 30, 4.0, 10000, 303, 200)
    table23_row(105, 4.0, 10000,  86,  70)
    table23_row( 40, 1.9, 10000, 250, 100)

    print "\t\\midrule"

    table23_row(  5, 3.6,  1000, 333, 300)
    table23_row( 10, 3.6,  1000, 111,  60)
    table23_row(  5, 1.9,  1000, 333, 300)
    table23_row( 15, 4.0,  1000,  66,  30)

    print "\t\\midrule"

    table23_row(  5, 3.6,   200,  57,  40)
    table23_row( 10, 4.0,   200,  20,  10)
```

```
    table23_row( 20, 1.9,   200,  12,  10)

def table23_row(H, G, S, R, w):
    r = run_table_row(w, None, 1, 1, 1, S/float(H*R), R, G, float(H))
    L_mean  = r['L'].mean()
    pF_mean = r['pF'].mean()*100
    pF_std  = r['pF'].std()*100
    pL_mean = r['pL'].mean()*100
    pL_std  = r['pL'].std()*100
    txt = ("\t{:.1f} & {:.2f} & {:.1f} & {} & {:.1f} & {:,} & {} & {} \\\\ "
           "% {:.3f} {:.3f}")
    print txt.format(L_mean, pF_mean, pL_mean, H, G, S, R, w, pF_std, pL_std)


def table32():
    print "%%%%%%%%%%%%%%%"
    print "% Table 3.2. %"
    print "%%%%%%%%%%%%%%%"
    print
              # C |D1|  S0   | H |  R |  G |  w  #
    table32_row(10, 4, 10000,  5, 125, 1.8, 2000)
    table32_row( 1, 4, 10000,  5, 384, 3.8,  400)
    table32_row( 4, 4, 10000,  5, 178, 3.2,  500)
    table32_row(10, 1, 10000,  5, 333, 3.8,  200)
    table32_row( 4, 1, 10000, 10, 357, 3.6,  300)
    table32_row( 1, 1, 10000, 30, 303, 4.0,  200)

    print "\t\\midrule"

    table32_row( 1, 1,  1000,  5, 285, 4.0,  200)
    table32_row( 4, 4,  1000,  5,  25, 1.9,  200)
    table32_row( 1, 4,  1000,  5,  83, 1.9,  500)
    table32_row( 4, 1,  1000,  5,  83, 3.8,   60)
    table32_row(10, 4,  1000,  5,  10, 1.8,   70)

    print "\t\\midrule"

    table32_row( 1, 1,   200,  5,  57, 3.8,   40)
    table32_row( 1, 4,   200,  5,  16, 1.8,   80)
    table32_row( 4, 1,   200,  5,  16, 3.8,   10)
    table32_row( 4, 4,   200,  5,   5, 1.9,   40)

def table32_row(C, D1, S, H, R, G, w):
    D2 = {1: 1, 4: 7}
    r = run_table_row(w, None, C, D1, D2[D1], S/float(H*R), R, G, float(H))
    L_mean  = r['L'].mean()
    pF_mean = r['pF'].mean()*100
```

```
    pF_std  = r['pF'].std()*100
    pL_mean = r['pL'].mean()*100
    pL_std  = r['pL'].std()*100
    txt = ("\t{} & {} & {:,} & {:.0f} & {} & {} & {:.1f} & {:.2f} & {:.2f} & {} & "
           "{:.2f} \\\\ % {:.3f} {:.3f}")
    print txt.format(C, D1, S, L_mean, H, R, G, pF_mean, pL_mean, w, 0,
                     pF_std, pL_std)


if __name__ == "__main__":
    table21()
    print
    print
    table23()
    print
    print
    table32()
```