# Investigation and Modelling of a Cortical Learning Algorithm in the Neocortex

Dobric, Damir

https://pearl.plymouth.ac.uk/handle/10026.1/22109

http://dx.doi.org/10.24382/5155
University of Plymouth

# Investigation and Modelling of a Cortical Learning Algorithm in the Neocortex

by

Damir Dobric

A thesis submitted to the University of Plymouth in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Engineering, Computing and Mathematics
June 2023

# Acknowledgements

I thank my Director of Studies, Prof. Dr Andreas Pech, who supported me on this journey. Also, thanks to my supervisors, Prof. Dr Bogdan Ghita and Prof. Dr Thomas Wennekers, for their help, sustainable feedback and support.

Also, many thanks to the university organisation's staff, who made it possible to continue the research remotely during COVID restrictions.

Last but not least, I sincerely thank my colleagues, friends and family, who helped me allocate the required focus time.

# Author's Declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without the prior agreement of the Doctoral College Quality Sub-Committee. Work submitted for this research degree at the University of Plymouth has not formed part of any other degree at the University or another establishment.
This work has been carried out by Damir Dobric under the supervision of Prof. Dr Andreas Pech, Prof. Dr Bogdan Ghita, and Prof. Dr Thomas Wennekers.

## Parts of the thesis that the author has published

Dobric, Pech, Ghita, Wennekers 2020. 2020 International Journal of Artificial Intelligence and Applications. *Scaling the HTM Spatial Pooler*. doi:10.5121/ijaia .2020.11407

Dobric, Pech, Ghita, Wennekers 2020. 2020 AIS 2020 - 6th International Conference on Artificial Intelligence and Soft Computing, Helsinki. *The Parallel HTM Spatial Pooler with Actor Model*. https://aircconline.com/csit/csit1006.pdf, doi:10.5121/csit.2020.100606

Dobric, Pech, Ghita, Wennekers 2020. 2020 Symposium on Pattern Recognition and Applications, Rome. *On the Relationship Between Input Sparsity and Noise Robustness in Hierarchical Temporal Memory Spatial Pooler*.
*h*ttps://dl.acm.org/doi/10.1145/3393822.3432317. doi:10.1145/3393822.3432317

Dobric, Pech, Ghita, Wennekers 2021. ICPRAM Vienna (Awarded as Best Industrial Paper).
*Improved HTM Spatial Pooler with Homeostatic Plasticity control*.
doi:10.5220/0010314200980106

Dobric, Pech, Ghita, Wennekers 2022. Springer Nature Computer Science Journal
*On the Importance of the Newborn Stage When Learning Patterns with the Spatial Pooler*.
https://rdcu.be/clcoc. doi:10.1007/s42979-022-01066-4

Word count for the main body of the thesis: 47769

Signed [illegible] Date: 10.06.2023

# Abstract

Title: *Investigation and Modelling of a Cortical Learning Algorithm in the Neocortex*
Damir Dobric

Many algorithms today provide a good machine learning solution in the specific problem domain, like pattern recognition, clustering, classification, sequence learning, image recognition, etc. They are all suitable for solving some particular problem but are limited regarding flexibility. For example, the algorithm that plays Go cannot do image classification, anomaly detection, or learn sequences. Inspired by the functioning of the neocortex, this work investigates if it is possible to design and implement a universal algorithm that can solve more complex tasks more intelligently in the way the neocortex does. Motivated by the remarkable replication degree of the same and similar circuitry structures in the entire neocortex, this work focuses on the idea of the generality of the neocortex cortical algorithm and suggests the existence of canonical cortical units that can solve more complex tasks if combined in the right way inside of a neural network. Unlike traditional neural networks, algorithms used and created in this work rely only on the finding of neural sciences. Initially inspired by the concept of Hierarchical Temporal Memory (HTM), this work demonstrates how Sparse Encoding, Spatial- and Sequence-Learning can be used to model an artificial cortical area with the cortical algorithm called *Neural Association Algorithm* (NAA). The proposed algorithm generalises the HTM and can form canonical units that consist of biologically inspired neurons, synapses, and dendrite segments and explains how interconnected canonical units can build a semantical meaning.

Results demonstrate how such units can store a large amount of information, learn sequences, build contextual associations that create meaning and provide robustness to noise with high spatial similarity. Inspired by findings in neurosciences, this work also improves some aspects of the existing HTM and introduces the *newborn* stage of the algorithm. The extended algorithm takes control of a homeostatic plasticity mechanism and ensures that learned patterns remain stable.

Finally, this work also delivers the algorithm for the computation over distributed mini-columns that can be executed in parallel using the Actor Programming Model.

# Contents

# 1 Introduction

Nowadays, artificial intelligence and machine learning are a megatrend successfully applied in the industry. Many techniques, algorithms, and approaches today, like linear regression, clustering, sequence learning, image recognition, and many others, solve common machine-learning problems. However, they are all explicitly designed to provide a solution in a specific problem domain. Even if successfully applied in the industry, in contrast to the brain, they are not flexible and cannot adapt and solve problems they are not designed for. For example, algorithms that learn sequences cannot do clustering, or a game-playing algorithm cannot do anomaly detection. A general motivation was to build a smart or intelligent machine. But, algorithms developed in many directions like supervised, unsupervised, or reinforcement learning. They use different approaches and can solve specific problems only. They are all successfully applied in the industry but are not intelligent. The brain was and still is the motivation of many studies. Some algorithms are inspired by the brain's functioning and incorporate some general properties of neurons and their populations. They build artificial neural networks and involve various learning rules driven by the psychological hypothesis that defines how synaptic connections change depending on the experienced pattern. Such algorithms can learn and process a lot of information, but they are still not intelligent and do not work as the brain does. This fact implies questions like "What is intelligence" "Is it possible to design an intelligent machine" and so on. Inspired by these questions, the original idea of this work was to investigate if and how it is possible to design and implement an algorithm that aligns with a hypothetical cortical algorithm. At the beginning of this work, many algorithms have been analysed to understand different learning techniques with different goals, approaches, and motivations. The book "On Intelligence" (Hawkins, Blakeslee, 2004), later followed by a second edition, "A thousand brains" (Hawkins, Dawkins, 2021), delivered a promising approach that sounded feasible.

The research in neurosciences (Ahmad, Hawkins, Cui, 2017) indicates that the neocortex might contain the cortical algorithm, which is universal and adaptable. Motivated by the remarkable replication degree and uniformity (Kaas, 2012), (Hubel, Wiesel, 1974) of the same and similar circuitry structures (Douglas, Martin, 1989) in the entire neocortex, this work focuses on the idea of the generality of the neocortex algorithms.

Furthermore, it suggests the hypothesis of canonical cortical units that can solve more complex tasks if combined in the right way inside a neural network.
This research investigates whether designing and implementing such an algorithm is feasible. The review undertaken in this research shows that algorithms like Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) best align with biological learning processes found in the neocortex. It is a new algorithm still under research with immense potential (Yuwei, Ahmad, Hawkins, 2017), (Wielgosz, Pietroń, Wiatr, 2016), (Melis, Chizuwa, Kameyama, 2009), (Bonhof, 2008), (Ahmad, Lavin, Purdy, 2017) etc.

## 1.1 The Contribution area

This section summarises the scientific and engineering contributions of this research. This work relies on many biological findings to create the artificial cortical learning algorithm. It explicitly avoids any approach or solution not previously found in biology. Most, but still not all, theoretical parts of this work find a corresponding implementation used for experimental work and delivered as an open-source framework.

This thesis's contributions target Computational Intelligence and Parallel Computing. The Computational Intelligence part starts with analysing the existing cortical learning algorithms and related findings in neural sciences. Then, it investigates the robustness to noise of the algorithm and analyses the similarity capability. It demonstrates how memorising spatial patterns can store a considerable amount of information and simultaneously be naturally robust to noise and build similarities.

Furthermore, this work extends and generalises the existing Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) and shows how to achieve stable learning by putting the algorithm in a state that corresponds *newborn* stage of mammals. Also, the new extended algorithm shows how to solve the problem called the stability-plasticity dilemma.

Finally, inspired by HTM CLA, it suggests a generalised cortical algorithm called *Neural Association Algorithm*.

The Parallel Computation part of this work analyses different approaches providing the solution for the parallel execution of the cortical algorithm in the cluster of many distributed nodes. Finally, it proposes the best method for parallelising described cortical algorithms.

## 1.2 Thesis Structure

The thesis thematically contains five parts. The first part is a *background* overview of several scientific areas related to this work. It analyses relevant findings in Neurosciences, Machine Learning, Cortical Algorithms, and Parallel Programming techniques. The second part, *Design and Implementation of the Cortical Algorithm*, describes how the cortical algorithm is designed and implemented. The third part of the thesis, *Parallel Computation, focuses on the* parallel design of the cortical area. Finally, part four, *Cortical Capabilities,* focuses on essential features of the cortical algorithm and provides some improvements, like a definition of the *newborn* stage of the cortical algorithm.

The last, fifth part, *Discussions,* gives a critical discussion of the results of this work, finalised by recommendations for future work.

### 1.2.1 Part one – The background

The first introductory part consists of chapters 2, 3, and 4. Chapter 2 introduces the biological background related to the functioning of the cortical learning algorithm. It summarises the most relevant findings in the field of neurosciences, which have considerable importance for research in this work. Described findings are used as a foundation for modelling the cortical algorithm.

Chapter 3 focuses on machine learning algorithms that mostly align with the biology of the *neocortex* described in chapter 2. It briefly explains the HTM cortical learning algorithm and how this research helps build the CLA in this work. Finally, this chapter summarises a few other biologically inspired algorithms analysed in this work.

Chapter 4 summarises, in the context of this work, the most important parallel programming techniques analysed to create the parallel version of the cortical algorithm, which is a focus of part three of this work.

### 1.2.2 Part two - Design and implementation of the Cortical Algorithm

Part two (chapter 5) describes the design of the cortical learning algorithm created in this work. It starts with explaining the enormous memorising capacity of the SDR encoding and briefly describes modelling the cortical area, including sensory encoding, activation of synapses, and dendrites. Next, it describes how HTM-inspired Spatial Pooler and Temporal Memory algorithms have been designed and implemented in this work. Finally, this part introduces a Neural Association Algorithm.

### 1.2.3 Part three – Parallel computation of the cortical area

Chapter 6 describes the concept of the parallel model of computation inside the Cortical Algorithm over mini-columns and a cortical area. The idea expressed in this chapter proposes a computation model based on the Actor Programming Model (see chapter 4). It explains the redesign of the open-source *neocortexapi* framework (Dobric, 2019) extended to support the Parallel Computation on the example of the Spatial Pooler. This chapter summarises the most critical findings that also have been published at the International Conference on Artificial Intelligence and Soft Computing (Dobric, Pech, Wennekers, Ghita, 2020) and in the International Journal for Artificial Intelligence (Dobric, Pech, Ghita, Wennekers, 2020). It also gives an idea of the model for general parallel execution of the entire cortical area.

### 1.2.4 Part four – Cortical Capabilities of the Neural Association Algorithm

Chapters 7 and 8 describe the essential capabilities of the cortical algorithm designed in part two and published as part of this research. Chapter 7 focuses the noise robustness and spatial similarity, which are features of the SP design. It describes how the sparsely encoded pattern is robust against noise and how spatial similarity is achieved. Chapter 8 improves the Spatial Pooler algorithm by introducing a *newborn* stage of the cortical algorithm. Inspired by findings in neurosciences, It shows how controlling homeostatic plasticity can enhance the learning process and clarifies why such an infant stage is essential for the cortical algorithm of species.

### 1.2.5 Part five – Discussions and future work

This part (chapter 9) summarises all results and provides a critical discussion on all topics. Finally, this part also includes a list of future work items needed to make the results of this work industry ready.

# 2 Biology of the Neocortex

Recent research studies in neurosciences suggest that the same principles for information processing inside the neocortex might be used to implement a computational system that aligns with the brain's biological architecture and functionality. The motivation of such systems is to provide more cognitive capabilities than standard neural networks and today's machine learning algorithms.

Most traditional machine learning algorithms rely on statistical analysis or a mathematical approach to solve specific problems (LeCun, Bengio, Hinton, 2015). Some find their roots in the biological processing of information by different brain parts but do not necessarily provide a fundament for cognitive capabilities. Instead, most solutions are math-focused, allowing them to solve specific problems. However, because of the dedicated focus on a particular problem domain, such algorithms are not applicable when it comes to flexibility or implementing cognitive features.

More than a hundred years of research on neurosciences seem to provide a tremendous amount of information today, which can be used to design and implement a cortical algorithm that requires less computation power and naturally provides cognitive features in the core of its structure. Most of the research summarised in this work relates to the neocortex because it is evolutionary, the "new part of the brain" probably responsible for intelligence (Hawkins, Ahmad, 2016).

The human brain contains approximately 100 billion neurons (Herculano-Houzel, 2009) and more than 100 billion glial cells (Solomon, Henry, Sushmitha, Partha, Melwin, 2015), the tissue surrounding the neurons, providing oxygen for neurons. Some studies assume that an increase in the number of neurons in several brain areas is concurrent with an increase in the number of non-neurons (i.e. glial cells). They show that the ratio of *glia/neurons* increases with brain size. In other words, the mass of the brain does not necessarily lead to higher cognitive capabilities because the number of neurons does not increase proportionally to group (Herculano-Houzel, 2009). Furthermore, different cellular rules in species show clearly that a higher mass does not imply a higher number of neurons. However, a higher number of neurons directly indicate increased cognitive capabilities.

The brain consists of different kinds of cells, which fulfil some functions and form the brain's tissue. However, not necessarily all of them play an essential role in developing cognitive features. For example, as known today, the *glia-cells* seem not to influence cognitive characteristics of the brain, even if their number is high compared to other kinds of neuronal cells. Other types of neurons are spiny and non-spiny neurons. Spiny neurons are mostly pyramidal (75-80% of all neurons) and found in mini-columns and play an essential role in this research. Non-spiny cells are so-called inhibitory cells (10-25%) of the neocortex.
The estimated number of 100 billion neurons represents a significant challenge for modelling a cortical algorithm on modern computers: according to different sources (Solomon D, Henri, Charles, Melwin, Ninoshka, 2015), a single neuron might have up to a hundred thousand synapses. Assuming that a single neuron forms one to ten thousand synapses to other neurons or input sensory cells, the human brain may contain more than 10E+14 synapses of ten thousand synapses per neuron.

The complexity of the brain is unlikely to be matched soon by the existing computing paradigms. Therefore, from a practical perspective, to represent the above synapses as an array on a modern software development framework used in this work, like the .NET Core 64-bit operative system's maximum possible array size of integers (32 bits) is *2,147,483,592*. Therefore, this number is calculated as follows:

$$\frac{2^{32}}{2} - 56 = 2147483592 \quad (1)$$

Half of the maximal integer value on a 64-bit system is subtracted by 56, an internal framework overhead to hold and manage an array. The solution will require approximately 50000 such arrays to achieve a hundred billion synapses,

$$\frac{10^{15}}{2147483591} = 46566.13 \quad (2)$$

The neocortex is a thin sheet of different kinds of neural cells (see Figure 1), which are highly interconnected and occupy a fragile surface area (Mountcastle, 1997). All neurons in the neocortex are organised into six horizontally layered columnar structures (Kaas, 2012).

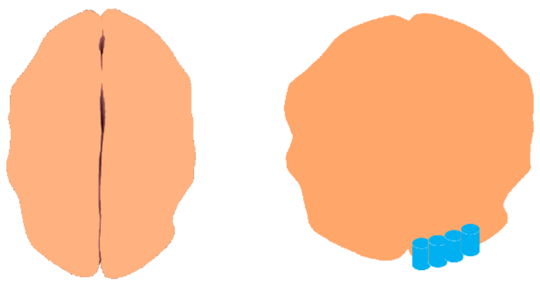


*Figure 1.*

*The neocortex is a thin surface area of approx. 2600 cm$^2$ with a thickness of 3–4 mm. On the left: neocortex unfolded tissue. On the right: unfolded neocortex tissue consisting of cortical cells (in blue) distributed vertically across the cortex sheet.*

Inside all layers, neurons are organised in columnar structures. One of the goals of neurosciences is to understand how the complex columnar organisation generates and controls human behaviour. The organisation of neurons seems to follow the same pattern across the entire neocortex, which plays a vital role in this work. They build a complex and highly interconnected "distributed system" (Mountcastle, 1997). It is assumed that there is a small canonical unit inside the population of neurons. This unit might be essential in solving basic and complex cognitive tasks. It is created from a population of cells and forms structures called mini-columns. Further, mini-columns form cortical columns, forming a more complex system called layers.

The following sections describe more precisely the most relevant entities of the brain and their features used for investigating the cortical algorithm and its parallel execution.

## 2.1 Neurons, synapses and dendrites

Reverse engineering of the neocortex or the brain itself is challenging. Nowadays, how neurons process the input from many synapses is still a mystery. They are one of the essential lower-level elements of neural biology in this context. Simplified, the neural cell (neuron) consists of the cell body (soma), dendrite segments, axons and synapses.

When the membrane potential of the neural cell rapidly oscillates, the neuron generates the action potential (Chen; Lui., 2021). The action potential brings the neuron to the firing or spiking state. A neuron "computes" input into a sequence of action potentials or spikes (Arcas, Fairhall, Bialek, 2003). The firing frequency of the neuron can vary and might encode even space and time (Hasselmo, 2013), which is more than a single bit of information. In this work, the spiking (firing) neuron represents the carrier of a single bit of information.

Axon is a part of the neuron that arises from the cell body. It transforms impulses from one neuron to one or more other neurons. The cell that sends the signal (fires) is called a presynaptic cell or sometimes, in this work, referred to as a source cell. Before the axon terminates, it spreads into branches ending with terminals that connect to postsynaptic neural cells. Axon terminals of the presynaptic cell end with synaptic connections to dendrites and dendritic spines of the postsynaptic cell. Receiving and processing the information in the neuron occurs in the dendrites and the cell body.
Neurons generally have an excitatory or inhibitory function (Swanson, Maffei, 2019). Information flows via excitatory neurons and excites the neuron to spike. In contrast, inhibitory neurons prevent a neuron from spiking. Inhibition and excitation are both critical and keep information processing and encoding balanced.
Synaptic connections between neurons influence the learning process in the brain, and there is evidence that they also form memories (Abraham, Jones, Glanzman, 2019). Synaptic connections are created, destroyed and remodelled (Chklovski, Mel, Svoboda, 2004).

Their ability to strengthen or weaken synaptic connectivity over time is called functional plasticity. The synaptic plasticity in this work is mainly based on the Hebb Rules (Hebb, 1949) encapsulated in the phrase "cells that fire together wire together".

## 2.2 Columns

One of the critical entities in the cortex is the classic column or cortical column, sometimes called a module (Mountcastle, 1997). This entity will be referred to in this work as a cortical column. It represents a grouping of cells across layers, which shares the same sensory input (see Figure 4). Different sensory inputs stimulate specific regions and are associated with a particular column. The size of one cortical column may vary from 300 μm to 600 μm in diameter.

Cortical columns are organised in groups and form another entity in this work called an area. Areas (in the HTM context, also called regions) are highly interconnected and build a hierarchical information processing system. It is assumed that every area is associated with a unique semantical level of processing. This assumption does not affect the cortical algorithm because it does not require any specific design and implementation detail. It is used solely to express that area A1 processes lower-level information (i.e., sensory input). A1 output is passed as input to the higher-level area A2, which will build higher-level information from pre-processed sensory input.
For example, visual information processing in the brain is done in areas V1, V2 and V3 (see Figure 2). Every area has a particular function in a cortical algorithm. According to many papers, i.e.: (Kaas, 2012) and (Lyon, 2001), these areas form a semantical hierarchical tree for processing information: V1->V2->V3, where the volume of areas satisfies V1 > V2 > V3.
In this example, information flows from the retina to V1 and V2 and V3. Every named area sends feedback information to the area lower in the hierarchy, which is the opposite of the feed-forward flow.

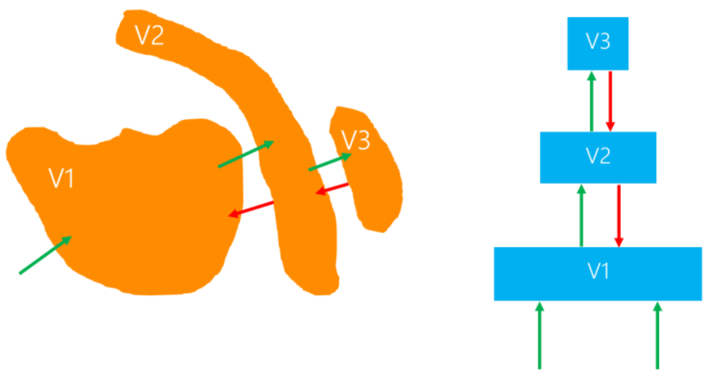


*Figure 2.*
*Shows information flow in areas of visual cortex V1, V2 and V3. The flow of information indicates hierarchical connectivity between tree areas. For example, information is received from the retina (not shown in the picture) and forwarded to V1, then V2 and finally to V3. Feedback flow is the opposite.*

As described later, this semantical hierarchical structure is one of the crucial features of the cortical algorithm. Furthermore, the hierarchical organisation implies the idea of the modular sensory cortex and the concept of canonical cortical circuits. Specifically, regarding processing information in V1 and V2, there is also a concept of the hyper column consisting of cells responding to different aspects of visual stimulus like detecting lines, their orientation and colour (Ts'o, Zarella, Burkitt, 2009).

Understanding the neocortex's semantical topologies is crucial to scaling the cortical algorithm. The organisation of cells in the region is essential for this work. Cells form a large number of synapses and interconnected areas (regions). A large number of connections impact parallel execution and the scaling algorithm, and it plays a vital role in building higher-level meaning based on information shared in semantically lower-level areas.

## 2.3 Mini-columns

The vertical cell column of the neocortex has been a model for a cortical organisation for some time (Mountcastle, 1957; Mountcastle, 1997). It is horizontally organised into six layers and vertically into a group of cells connected by synapses across layers (see Figure 3). The well-structured horizontal population of neurons inside the column forms another important entity called mini-columns. Mini-columns have been developed during the evolution of all mammalian brains. Researchers believe that the brain cortex increases by adding new elementary units (mini-columns) rather than simply increasing the number of neurons. The estimated diameter of a single mini-column inside the *cortical column* falls within 30–50 µm (Buxhoeveden, Casanova, 2002). Mini-columns span multiple layers and typically share the same proximal receptive field properties.

The columnar structure in the neocortex and synaptic connections follow the concept of a biologically distributed system. In this context, a distributed system is defined as a set of spatially separate "units" that communicate by exchanging messages by firing neurons.

The neocortex contains two basic types of neurons: neurons spiny and none-spiny (Buxhoeveden, Casanova, 2002). Spiny (stellate) neurons are mostly pyramidal (75-80% of all neurons) and found in mini-columns. They play an essential role in this work because they are excitatory cells carrying information across neurons. Non-spiny cells occupy 10-25% of the neocortex. In contrast to spiny cells, they are inhibitory cells.

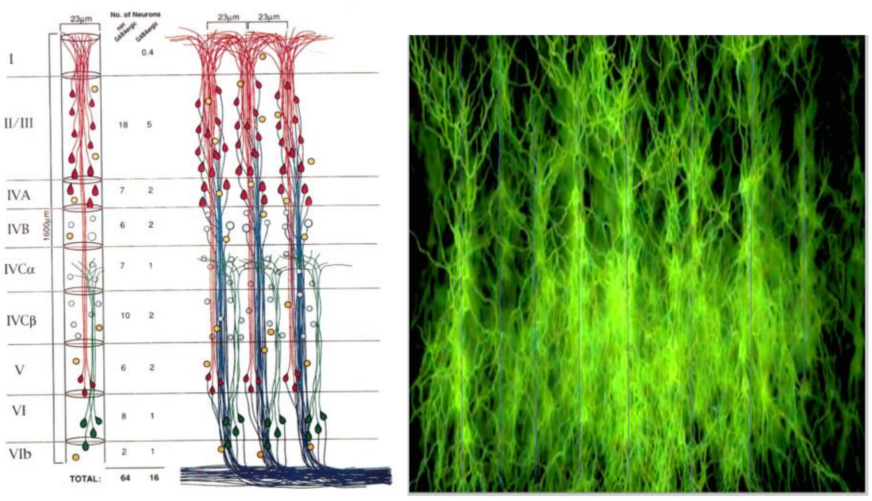


*Figure 3*
*The figure shows the organization of mini-columns inside of a cortical column. Mini-columns typically span all layers. Left: An arrangement of different types of neurons in the mini-column across layers (Mountcastle, 1997). Right: Thin vertical lines in the simulation show the high cellular concentration of different cells (depending on the layer), forming mini-columns.*

## 2.4 Layers

One of the brain's uniqueness is the layered structure that might play an essential role in the brain's functioning (Guy, Staiger, 2017). Layers are sheets of populations of neurons layered on top of each other that share some commonalities. Brains and parts of the brain might generally have a variable number of layers. For example, the human neocortex is divided into six layers (Mountcastle, 1997). In contrast, the three-layer cortex arose in the early amniote forebrain's olfactory, hippocampal and dorsal cortex. Three-layer and six-layer cortexes are built on pyramidal cells with proximal, apical and basal (distal) dendritic trees (Shepherd, 2011). It is possible to identify basic circuits for recurrent excitation and lateral inhibition across all the cortical regions. It seems that the layer-specific function does not reside in the layer but in the circuit, irrespective of its topology (Guy, Staiger, 2017).

Every layer is typically formed by many cells of a few different cell types. For example, every cortical column in the area represents an information processing unit whose neural cells are layered (see Figure 4).

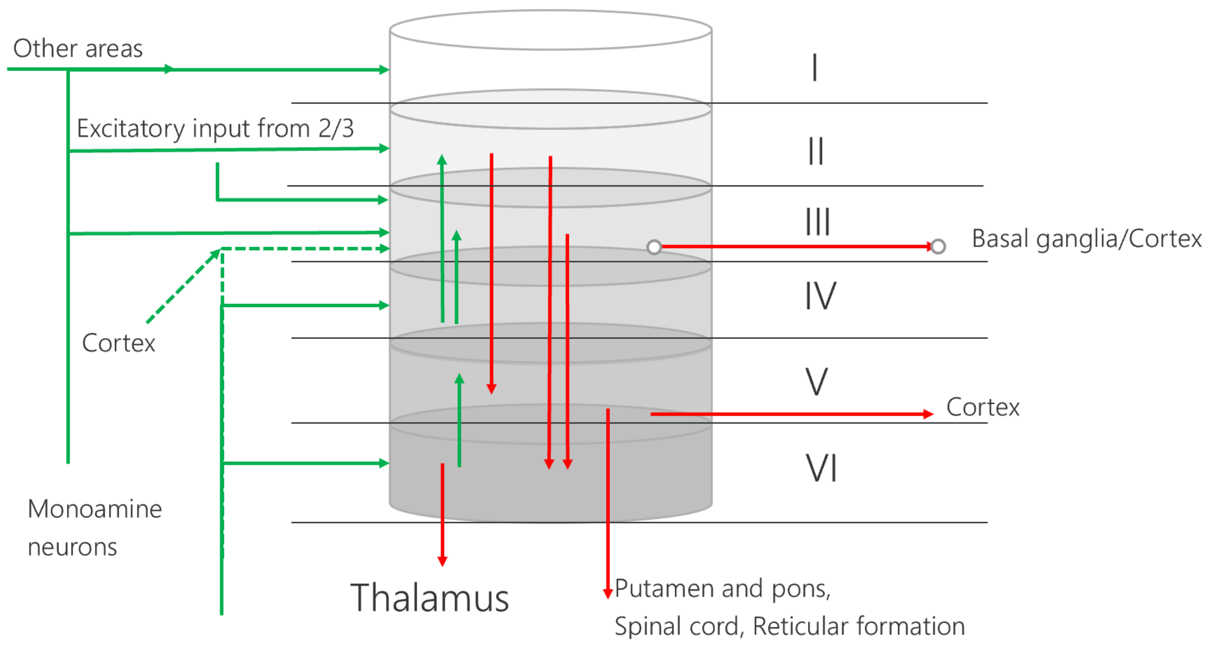


*Figure 4*

*They layered cellular organisation. Every cortical column spans six layers and spreads across a 300 to 600 µm diameter range. All layers have different functions in the brain. For example, layer IV consists of stellate and pyramidal neurons, which receive a feed-forward sensory input. Layers II/III (usually considered together) are responsible for sequence learning.*

The function and interaction within each layer are as follows:

Layer I - receives feedback axons from higher-level areas and the central nervous system (usually monoamine neurons). These connections receive feedback axons from higher levels in the information processing hierarchy. (Thomson, 2003).

Layers II and III – are the so-called output layer, which provides temporal information processing and results in higher-level projection. These layers might be responsible for learning sequences (Hawkins, Ahmad, 2016).

Layer IV – receives the feed-forward input, not involved in receiving feedback information. Also, it sends axons to other areas and forms long-distance horizontal connections within the layer (Douglas, Martin, 2004). This layer plays a major role in spatial sparse pattern representation. Most of this work's investigation and modelling of the cortical algorithm is related to this layer.

Layer V – it has been suggested that this layer encodes the duration between sequences (George, Hawkins, 2009).

Layer VI - is connected to the thalamus, sending processing feedback to lower layers in the hierarchy, receiving feed-forward input, and sending its output to layer V (Hawkins, Ahmad, Subutai, Yuwei, 2017). Layers VI and V, similarly to layers IV and II/II, provide an input-output circuit instance, which can be modelled as a broadly distributed neocortex.

Making the algorithm that models (emulates) this layering structure is vital for this work. Layers are still under research, and today it is unknown how layers exactly should or can be interconnected to achieve the best cognitive capabilities. Following the idea of the canonical microcircuit in the visual cortex (Hubel, Wiesel, 1974), (Douglas, Martin, 1989) and the hypothesis of the possible existence of the canonical unit that can solve some higher cognitive tasks. This work uses the layered structure approach (among others) to model the cortical algorithm. In this work, layers host smaller units that incorporate algorithms for spatial recognition, associative learning and sensory encoding.

## 2.5 Dendrites

Excitatory input of neurons typically comes over dendrite segments. The excitation subthreshold voltage is directly dependent on the relative location of the dendrite to the cell body (Williams, 2005). The dendrites with an approximate distance from the cell body of 250 µm are called proximal dendrites. Dendrites with an approximate distance of 500 µm belong to the distal (basal) dendrite segments category. The excitation voltage of the neuron decreases exponentially with the distance of the dendrite from the soma.

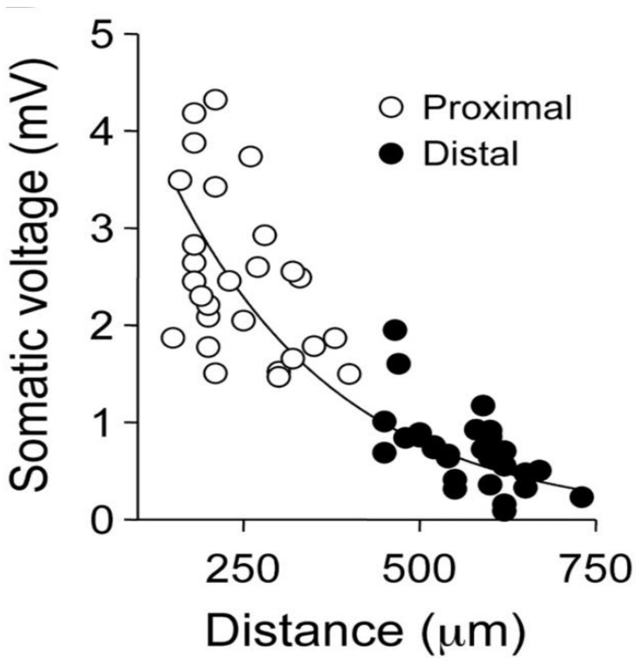


*Figure 5*

*The influence of the distance of dendrites on the somatic voltage. Dendrites with a 250 µm distance (proximal) have a much higher impact on the polarization of the neuron (Williams, 2005) than dendrites with a distance larger than 500 µm (Distal)*

A similar finding (Larkum, Nevian, 2008) points out that the depolarization amplitude caused by the synaptic input is also a function of the distance from the soma. The local input resistance primarily determines this function.

Proximal dendrites are responsible for receiving feedforward input (Hawkins, Ahmad, 2016). The short distance to the soma and lower resistance considerably affect the cell and can directly generate the action potential.

Basal (distal) dendrites originate from nearby cells of the same region. They do not have a significant direct effect on the activation of the cell to an action potential. But they generate the NMDA spike (Larkum, Nevian, 2008), which can excite the cell and put it in the pre-depolarisation state. In this state, the cell is partially activated (depolarised). Therefore, clustered and multiple synchronous inputs to basal dendrites can have a more significant impact on a local dendritic spike than a somatic action potential. Dendritic signalling and information clustering are fundamental artefacts for understanding information processing in the brain.

Finally, apical dendrite segments are located far away from the cell's body (more than 1mm). They seem responsible for connecting different regions (i.e. higher-level regions). The NMDA spike generated in the apical dendrites does not directly affect the cell but leads to a Ca2+ spike. The exact correlation between NMDA spike, Ca2+ spike and action potential is still an area of research in neurosciences. Independent of how this biologically works, the feedback from higher-level regions can also lead to the cell's depolarization.

## 2.6 Encoding and the information persistence

One of the crucial questions to understand the information processing in the brain is to determine how neurons in a given area are activated and how this information is encoded. An important belief is that knowledge is coded in a distributed manner in the brain. This is also known as the Parallel Distributed Processing (PDP) theory of cognition (Bowers, 2009), also known as the dense coding scheme. In contrast, there is also a belief that knowledge is persisted in a localist fashion, where an object, some concept or even behaviour (cat, house, grandmother smiling etc.) is encoded by a hypothetically single neuron commonly called the grandmother neuron. This hypothetical neuron (Solomon, Henry, Sushmitha, Partha, Melwin, 2015) is also known as a Jennifer Aniston-Cell, Cardinal-Cell or a Gnostic-Cell. This theory is often titled a localist theory (Bowers, 2009). The key difference between the two theories is that each cell in localist theory codes precisely one item, and the PDP cell is involved in coding multiple items.

Hubel and Wiesel conducted experiments on the visual system of cats and monkeys, providing the first insights into the neural representations of visual information in the brain (Hubel, Wiesel, 1959). They detected firing neurons in the visual cortex when a line pattern was projected on a particular place on the retina depending on orientation or when the line was moving. They also showed the topological map of the visual cortex describing the columnar organisation of cells (Hubel, Wiesel, 1962). They explained how the V1 is organised in columns and holds so-called simple cells in coding similar but slightly different line orientations at the same retinal position (cells in adjacent columns code for line segments that vary by approximately 10° in their orientation). These columns are organised into hypercolumns and code for a range of line orientations in a range of retinal locations. Very important in this context is that information is encoded hierarchically. Complex cells in V1 are interconnected with multiple simple cells and encode more complex information (Hubel, Wiesel, 1968), which might support the evidence of grandmother neurons. These organised units encode visual information and pass them to higher-level regions of the brain, where the visual image is formed, and the model of the world is created. However, it is probably unrealistic to expect every mental state to be described by a single neuron (Hubel, 1995). Hubel and Wiesel's experiments clearly show that a minimal number of neurons fire when more complex moving shapes are detected. This result might be an indicator of evidence of sparse coding of information. However, Bowers (Bowers, 2009) described distributed codes as a representation in which each neuron is involved in coding more than one familiar "thing", and consequently, the identity of a stimulus cannot be determined from the activation of a single unit. He also argued that there is no unique coding scheme and pointed out Dense Distributed Representation (DDR) in relation to PDP, where each neuron is involved in coding many things. Another attractive coding scheme might be Coarse Coding. Multiple neurons are coactively firing in this coding mechanism when coding some item. In this case, neurons that encode similar things are located in neighbouring areas of the cortex.

The number of neurons and columns in the cortex has grown over the course of the evolution of the mammalian brain. However, it is surprising that many neocortical neurons show low firing rates on a stimulus. The experimental evidence (Barth, Poulet, 2012) in the mammalian neocortex indicates that most neurons keep silent (no firing), and only a tiny portion keep firing. This sparse encoding is achieved by competition across neurons and tuned by a homeostatic mechanism (Perrinet, Laurent, 2010), (Turrigiano, Nelson, 2004). Homeostatic signalling constrains neural plasticity and contributes to the uniform stability of neural function in the long term (Graeme. Davis, 2006), (Davis, 2013),The functional stability of neural circuits is achieved by homeostatic plasticity. It balances the network excitation and inhibition and coordinates changes in circuit connectivity (Tien and Kerschensteiner, 2018).

According to (Turrigiano, Nelson, 2004), the homeostatic plasticity boosting mechanism is only active in the early stage of the development of a *newborn* animal. Later, over time, the bosting is deactivated or shifted from cortical layer L4. This could mean that homeostatic plasticity in L4 has a task to uniformly activate mini-columns by continuously learning unsupervised patterns observed from the environment. However, continuous activation (boosting) changes synaptic weights and implies deleting previously learned patterns. This finding plays a vital role in this work in implementing the cortical learning algorithm, as briefly described in chapter 8. In addition to homeostatic plasticity, the adult mammalian cortex supports structural plasticity (structural dynamic) manifested by forming new synapses and destroying existing synapses (Zito, Svoboda, 2002). This process is dependent on genetic and environmental input influence.

The evidence of homeostatic plasticity correlates with another coding called Sparse Distributed Representation (SDR). The SDR describes complex stimuli with few neurons (Bowers, 2009). SDR enables high memory capacity and low overlap between stored information. Moreover, the relatively low number of neurons enables fast information processing compared to other coding mechanisms. Investigation of the functional significance of coding strategies provides many valuable insights, but the exact coding in the brain remains unclear. This is because it is generally challenging to make precise experimental observations in conjunction with many cellular dependencies (Finelli, Haney, Bazhenov, Stopfer, Sejnowski, 2007).

The synapses seem to exhibit a degree of stochasticity due to various noise sources, such as thermal fluctuations and variability in the release of neurotransmitters (Faisal, Selen, Wolpert, 2008). However, stochasticity does not necessarily require synaptic precision, which is the ability of synapses to transmit signals with high accuracy and reliability. This contradicts some traditional views in neuroscience, emphasising the importance of precise synaptic transmission for neural computation.

Some findings (Kajić, Gosmann, Stewart, Wennekers, Eliasmith, 2017) related to dendritic spikes suggested clustering of inputs along a part of dendritic segments. This may enable dendrites to separate threshold groups of similar inputs and allow a single neuron to recognise multiple patterns.

The current evidence suggests that the brain uses distributed codes in primary sensory areas and sparser and invariant representations in higher areas (Quiroga, Kreiman, 2010).

Another crucial biological aspect in this context is sensory encoding. For example, the ear audio signal is processed differently than the retina video signal. But, in both cases, some SDR is created and forwarded to the neocortex. The sensory encoding follows the principle explained in the following example. The cochlea in the ear is a mechanism for converting frequencies and amplitudes of sounds (Harold, Kirchner, 1974). The structure of a hair cell is organised in an array. Every cell in the array encodes a range of frequencies. Neighbourhood cells in this structure overlap their encoding frequency range. This implies two crucial claims. *First*, the overlap of the frequency range is a biological redundancy mechanism. If some cells are damaged, the neighbourhood cells will take over the encoding of frequencies owned by damaged cells. *Second*, similar frequencies are encoded similarly. That means that two nearby frequencies sound almost equally.

# 3 Machine Learning and Cortical Algorithms

At the beginning of this work, many algorithms have been analysed to understand different learning techniques and to find the best computational intelligence approach. The analysis was originally not only focused on brain-inspired algorithms. For example, traditional Machine Learning algorithms like Logistic Regression Restricted Boltzmann Machine, K-Means etc., were also analysed. However, algorithms like Perceptron, Multilayer Perceptron (MLP), Convolutional Networks, and others were more interesting for this work. This section summarises the most relevant findings related to this work.
Artificial Neural Networks (ANNs) are artificial adaptive systems inspired by the human brain (McCulloch, Pitts, 1943). They typically build a synthetic model of the biological neural network that contains mainly neurons and synapses. A typical learning method in this context is supervised learning that involves weight updates of synaptic connections using the backpropagation algorithm (Guo, Xiang, Zhang, Su, 2021). The backpropagation algorithm is efficient but still a mathematical construct without biological evidence. It updates the synaptic weights to narrow the output value to the expected target value. Unlike supervised learning, unsupervised learning is also a widely used algorithm (LeCun, Bengio, Hinton, 2015). A typical unsupervised learning method is based on Hebb-Rules (Hebb, 1949) and its spike-motivated temporally asymmetric form called Spike Timing Dependent Plasticity – STDP (Gerstner, Kempter, Hemmen, Wagner, 1996). The synaptic weight change Δwj of a presynaptic neuron j depends on the relative timing between presynaptic spike arrivals and dependent postsynaptic spikes. For example, suppose the presynaptic spike is repeated a few milliseconds before postsynaptic action potentials cause Long-Term Potentiation (LTP) of many synapse types. In contrast, if the presynaptic spike is repeated after postsynaptic spikes, the Long-Term Depression (LTD) of the synapse is caused.

As described later, the model investigated in this work is mainly based on the principles of Hebbian-Rules (Hebb, 1949). Following this rule, synaptic weight (permanence) is incremented by repeating input and decremented if the input doesn't stimulate the synapse. Using these simple rules synapses, durably persist the learned information in the form of weight. However, continuous learning naturally creates a constraint called the stability-plasticity dilemma. Learning requires plasticity for storing new knowledge and stability to prevent forgetting previous knowledge. Too much plasticity will result in forgetting, whereas too much stability will prevent learning (Martial, Aurélia, Patrick, 2013).

Merging HTM Hebb-Rules and STDP is a challenging task (Sebastian Billaudelle, Subutai Ahmad, 2016). Adopting the learning rules in HTM to the dynamic spiking model like HMF (Hybrid Multi-Scale Facility – University Heidelberg) is a nontrivial task. The cortical model investigated in this work does not integrate the complete chemical temporal asymmetry as suggested by STDP. Fortunately, the dynamic of synaptic activation in this work incorporates the resulting causality principle offered by STDP.

One of the prominent examples of brain-inspired algorithms solving classification problems is Multilayer Perceptron (MLP) and Deep Neural Networks (DNN). Such algorithms build a network (see Figure 6) of neurons connected by synapses across multiple layers (Arora, Basu, Mianjy, Mukherjee, 2018).

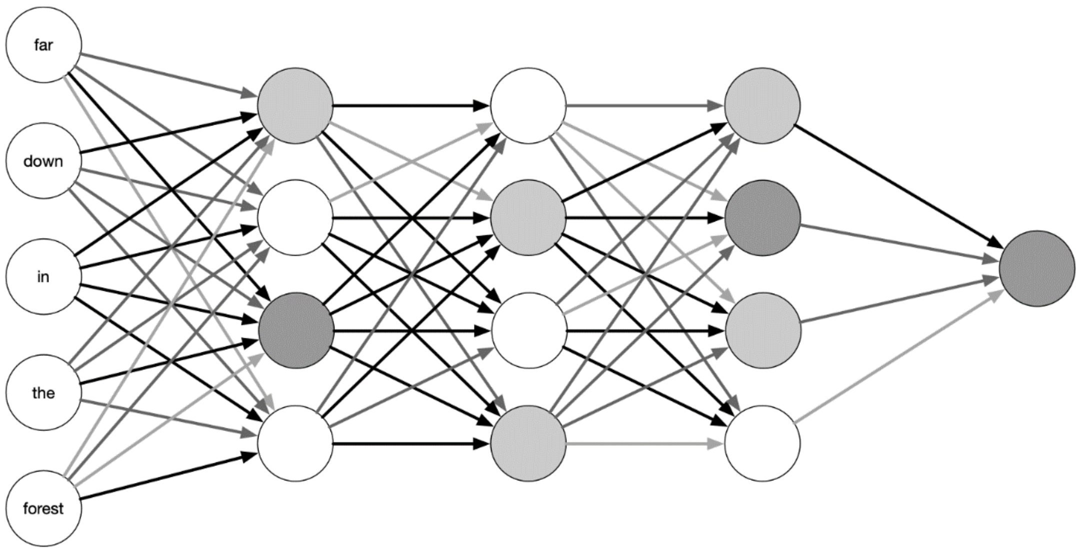


*Figure 6*
*Deep Neural Network model*

A popular deep learning algorithm inspired by the processing of visual information in the brain is a Convolutional Neural Network (Fukushima, 1980), initially called neocognitron. It is widely used for pattern recognition unaffected by a shift in position. This algorithm uses a Spatial Pooling and dimension reduction algorithm that typically performs a down-sampling of images. However, even if very efficient and biology-motivated, CNN does not work human-like. Moreover, some experiments challenge the plausibility of CNN as a theoretical framework for understanding image classification (Dietmar Heinke, Peter Wachman, Wieske von Zoest, E. Charles Leek, 2021).

Another important algorithm used for sequence learning in this context is the Recurring Neural Network (RNN) in its novel form, Long Short-Term Memory, initially introduced in 1991 (Hochreiter, Schmidhuber, 1997). This algorithm form is shown in Figure 7. It also uses neural cells interconnected with synapses (Sherstinsky, 2020).

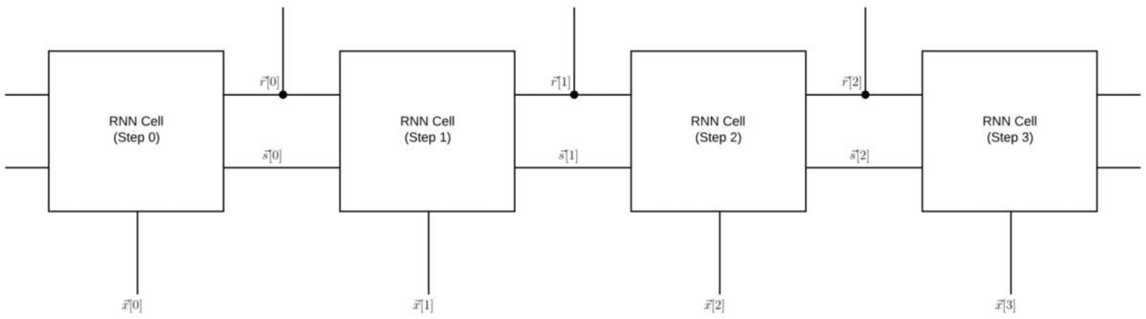


*Figure 7*
*Cell connections in a Recurring Neural Networks - Long Short-Term Memory (LSTM)*

All named algorithms perform well by solving dedicated problems in the industry. But they do not implement the cortical algorithm. They are brain-inspired and create some kind of neural network, but they all end up with a strong mathematical approach. The fact is, there is no evidence of such a robust mathematical formulation in the brain. The goal of this work is not to create just another better-performing mathematical formulation of the same approach. Instead, this work investigates whether canonical cortical units exist and how they might work. Some of the truly brain-inspired models described later in this chapter appeared to be *Ganger's* and *Hinton's Joint View* and *Object Models*. Also, the *DeepLearbra* framework provides an exciting approach in the context of the processing of visual streams. Because the idea of the HTM best aligns with biological findings described in the next chapter, this model plays the most critical role in this research.

## 3.1 Hierarchical Temporal Memory Cortical Algorithm

The Hierarchical Temporal Memory cortical learning algorithm (HTM-CLA) is a novel machine-learning algorithm inspired by the working principle of the neocortex, a part of the human brain believed to be responsible for intelligence (Hawkins, Ahmad, Dubinsky, 2011), (Hawkins, Dawkins, 2021). HTM-CLA, later in this document, also referred to as HTM, provides a theoretical framework that models several fundamental computational principles of the neocortex. The idea of the HTM-CLA was initially described in the book “On Intelligence” (Hawkins, Blakeslee, 2004), later followed by a second edition, “A thousand brains” (Hawkins, Dawkins, 2021). Today, the HTM idea is still in research. Still, the emerging framework already has many applications described in many papers (Bonhof, 2008), (Melis, Chizuwa, Kameyama, 2009), (Guo, Xie, Zhang, Li, 2015), (Shah, Ghate, Paranjape, Kumar, 2017), (Shah, Ghate, Paranjape, Kumar, 2018) , (Pal, Bhattacharya, Dey, Mukherjee, 2018), (Sousa, Lima, Abelha, Machado, 2021) etc. The core of the framework integrates several algorithms inspired by the neocortex’s functioning. As described later in this chapter, in contrast to DNN, the HTM uses a different, more complex biological model of the neuron (described later in chapter 5). It does not use backpropagation or any mathematical calculation as long there is no evidence that they are a property of the brain. Instead, it uses Hebbian learning with a natural STDP integration. This section gives a short introduction to HTM-CLA. A detailed description of the HTM-CLA, its adaption and implementation related to this work is presented in chapter 5.

The HTM-CLA introduces an artificial neuron model that is more complex than commonly used neuron models in neural networks. The neuron model implements the dendritic properties of pyramidal cells found in the neocortex (Spruston, 2008).
For example, a neuron in HTM can take a pre-depolarised state (see 2.5) in addition to active and inactive states. This kind of neuron state is suggested to be a *predictive state* (partially depolarised) of the cell. If the neuron is in a *predictive state*, it will probably generate an action potential (it will fire). Cells in this state fire earlier than cells stimulated only by feedforward stimulus and inhibit neighbourhood cells, creating even sparser pattern recognition. The HTM model hypothesises that active firing neurons correspond to bursting activity, and predictive state corresponds to tonic firing (Ferrier, 2014). In this state, the neuron remains depolarized, which is just below the threshold for firing an action potential.

Also, synaptic connections between cells and sensory inputs are more aligned with biology and use the concept of the receptive field (Lücke, 2004), (Lücke, Bouecke, 2005) instead of interconnecting all neurons across layers. The *permanence* value *'p'* (weight) defines the synaptic connection between the presynaptic cell and the dendrite segment of the postsynaptic cell.

$$\mathrm{p} \in \mathcal{R} \mid \ 0 \leq \mathrm{p} \leq 1 \qquad (3)$$

Every time some input pattern occurs, the permanence value increases by *permanence increment*, typically set at 0.15. An existing synaptic connection decrements if the spatial pattern does not appear. With this rule, synapses are formed, destroyed and strengthened. If the permanence of the synapse is above the threshold value *Permanence Connected*, the synapse is declared as connected. This value defines the synaptic binary state used in the HTM (Hawkins, Ahmad, 2016). With this rule, the HTM can learn and forget patterns (Quiroga, 2017). This process of synaptic connections becoming stronger with frequent activation is also called Long-Term Potentiation (LTP).

The HTM idea, in a nutshell, enables the creation of a cortical area (HTM region) or the whole cortical column (see 2.2), with any number of mini-columns and cells inside mini-columns. Most experiments in this work and papers related to HTM CLA are configured to run with 1024, 2048, and 4096 (Pietron, Wielgosz, Wiatr, 2016) and 16384 mini-columns. Figure 8 represents a simple HTM layer as a cortical area with two crucial algorithms, the Spatial Pooler and Temporal memory. The simplified layer model in the figure is built with six mini-columns with five cells each. As described later, models used in this work with 1024 or 2048 mini-columns demonstrate some cognitive capabilities.

Unlike standard neural networks, the HTM operates by design explicitly and exclusively on binary values ('0' and '1'). If the sensory input outside of the HTM area does not provide a binary input, the HTM offers the concept of encoders, which encode non-binary input in the required bit sequence. The primary information representation of the internally generated results in HTM is Sparse Distributed Representation (SDR), as described in more detail in the next section.

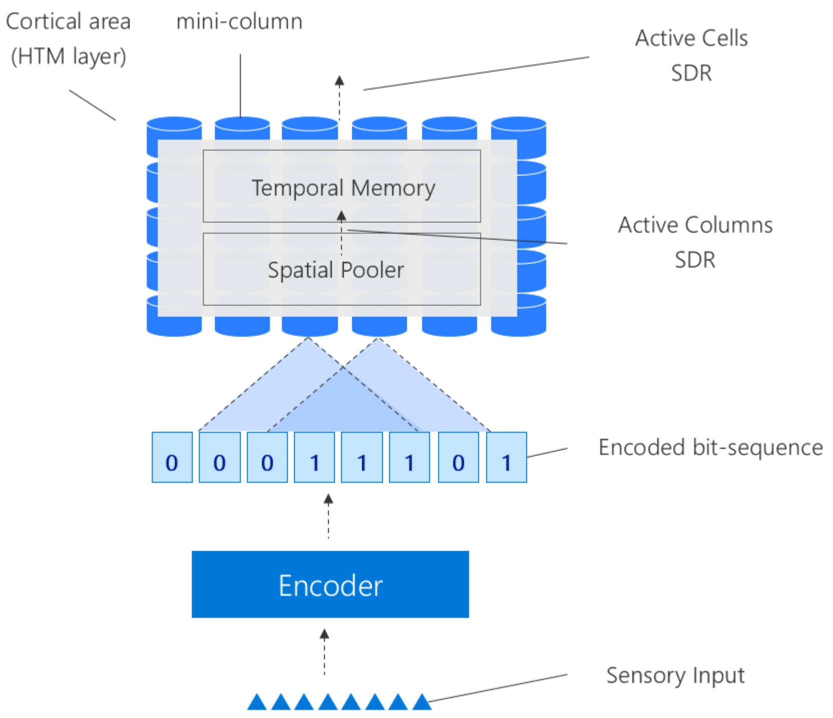


*Figure 8*

*HTML-CLA model with the encoder of sensory data and integrated Spatial Pooler and Temporal memory algorithm in a single cortical area.*

After encoding the sensory input (if required), the input bit-sequence is passed to the proximal dendrite segment of the HTM region, where the Spatial Pooler starts to learn the spatial input and encode it into the sparse code. The sparse code produced by the SP is encoded as a set of active mini-columns. The encoded sensory input (active mini-columns) is the input for the Temporal Memory algorithm, which is responsible for the learning sequences. The output of the TM is also a sparse code. The TM output is represented as a set of active cells.

The cortical area design flexibility of the HTM makes it possible to create any topology of the artificial tissue of the neocortex or brain. With this, minimal cortical areas can be created, and their cognitive capabilities can be investigated.
For example, the Temporal Memory algorithm (cell SDR) output can be further used as input for higher-level HTM areas (see Figure 9).

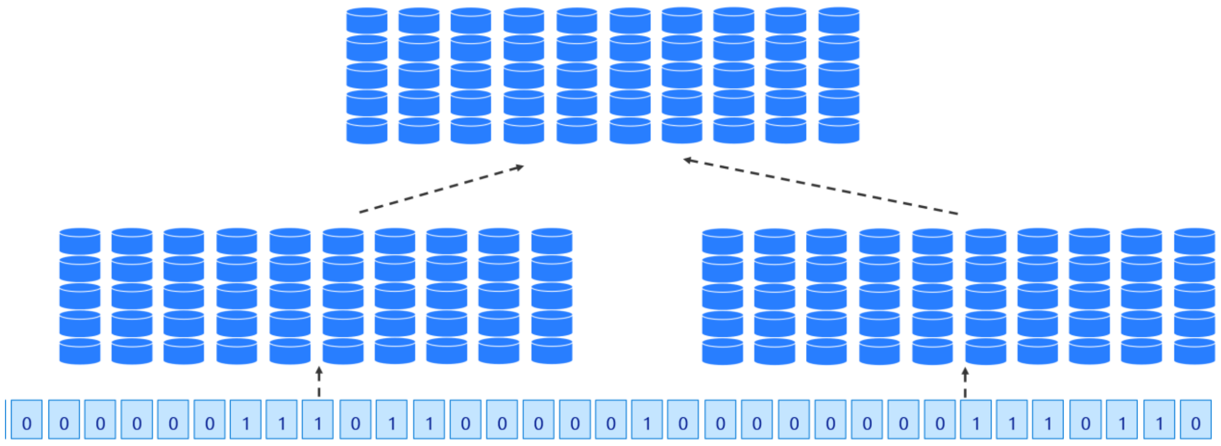


*Figure 9*

*The cortical area built with Hierarchical Temporal Memory with two lower layers and a single top layer.*

The first implementation of the HTM was published under the name *Nupic* (Numenta, 2008), followed by a new version in 2013. Shortly after, the open-source implementation of the JAVA HTM-CLA followed (open-source, htm-java, 2013). For this work, the HTM-CLA .NET Core C# version was implemented and published under the name *neocortexapi* (Dobric, 2019).

### 3.1.1 Sparse Representation of patterns

The synaptic mechanism in the brain, which drives neuronal circuit function while processing information, is still not entirely understood. As described in 2.6, the data can be encoded as dense, coarse or sparse. In a sparsity case, the information is encoded with typically less than 5% of active neurons in the population of some cortical area (Weliky, Fiser, Hunt, Wagner, 2003), (Hromádka, Weese, Zador, 2008). The HTM, in general, makes use of sparse encoding (Subutai, Hawkins, 2016). That means only a tiny fraction of neurons get active (see 2.6) and drive action potentials (Finelli, Haney, Bazhenov, Stopfer, Sejnowski, 2007) when the recognised pattern is encoded. For example, in a population of 100000 neurons, only 2% of neurons (2000) would become active (depolarised) if some pattern is recognised. The HTM defines the sparsity by ratio and can vary from a dense encoding scheme defined by PDP to a sparse representation (see 2.6). The effect of sparsity in HTM is that the cortical representation activates a tiny percentage of the neurons' populations while most remaining neurons stay inactive. This is a very efficient technique for memorising a large number of patterns inside a relatively small number of neurons (Larkum, Nevian, 2008). In contrast, neurons in typical ANN store a single pattern, known as a "*point neuron*".

Sparsity in this context is defined as ratio $a/n$, where $n$ is the number of neurons in the population, and $a$ is the number of neurons used to represent an encoded pattern. The population of neurons in HTM is considered to be sparse if a ≪ n. Sparseness or density at the time ***t*** is defined as a ratio of active neurons divided by the number of available neurons:

$$\boldsymbol{S_t} = \frac{1}{N}\sum_{k=1}^{N} a_{kt} \qquad (4)$$

$a_{kt}$ is a *k*th bit (neuron) at the time (step) $t$.

Given a number of bits in the SDR vector $n$ and the number of ON-bits $a$ that represents a pattern, the possible number of memorised patterns (capacity of the population) is calculated as:

$$\binom{n}{a} = \frac{n!}{a!\,(n-a)!} \qquad (5)$$

In the context of machine learning and pattern recognition, the number of recognised patterns is a subset of the set of all possible patterns. Given a $\theta$ as a threshold that splits recognised from unrecognised patterns, the number of possible recognised patterns is calculated as:

$$c_\theta = \sum_{\boldsymbol{i}=\theta}^{a} \binom{a}{i} \mid \theta < a \qquad (6)$$

However, the cell population also recognises many other patterns simultaneously, which might falsely activate a neuron. The possible activation set of all combinations is calculated as follows:

$$\Omega = \sum_{\boldsymbol{i}=\theta}^{a} \binom{a}{i} * \binom{n-a}{a-i} \qquad (7)$$

The probability of false positives $e$ can be calculated as Ω divided by the total capacity of the population:

$$e = \frac{\sum_{i=\theta}^{a} \binom{a}{i} * \binom{n-a}{a-i}}{\binom{n}{a}} \tag{8}$$

These equations will be used later in this document to describe how the HTM was modelled and implemented in the framework used in this work.

In general, sparse encoding algorithms originated in signal processing but are now extensively used in learning algorithms. From that perspective, they are related to Compressed Sensing reconstruction, also called CS-Theory (Donoho, 2006). The theory suggests that if the signal is sparse or compressive, then the original signal can be reconstructed. Sparse algorithms can be categorized from various viewpoints depending on their motivation, strategies, concerns, and so on. One interesting comparison and brief overview (Zhang, Xu, Yang, Li, Zhang, 2015) empirically categorize sparse encoding algorithms into four groups: greedy strategy, constrained optimization strategy, proximity algorithms, and homotropy algorithms. The same study categorizes sparse encoding algorithms based on norm minimization with respect to sparsity constraints: L0-norm minimization, Lp-norm ($0<p<1$) minimization, L1-norm, L2,1-norm and L2-norm minimization. It was also shown that the L2-norm is not strictly sparse. The study clearly shows that all sparse algorithms are rooted in various mathematical strategies used for the theoretical optimization of sparse coding.

This work, however, relies rather on biological than mathematical sparse representation algorithms. The biological motivation behind the theory of efficient information encoding belongs to computational theories of representational learning, and it is an important field of research. Some algorithms specifically suggest that efficiency in visual neuronal encoding implies that neurons sense independent shapes that constitute an image. For example, the causal generative image representation model called SparseNet (Olshausen, Field, 1996) uses Gabor filters to create a localized, oriented receptive field similar to that in the primary visual cortex described by Hubel and Wiesel in 2.6. Similarly, (Bell, Sejnowsky, 1997) developed a sparse encoding algorithm based on Independent Component Analysis (ICA). Both models generate smooth neural activity with a smaller variance (narrower or peakier) Gaussian distribution.

The more efficient form of sparse encoding provides the model Sparse-Set Coding network (SSC). In this encoding mechanism, neurons and mini-columns can only take binary states (Rhen, Sommer, 2006). Another work (Thronton, Srbic, Main, Chitsaz, 2011) uses sparse encoding to encode the input by setting the synaptic permanences to an integer value rather than using binary values. Synapses used in the learning process are created between mini-columns and input neurons. This technique might potentially be suitable for encoding images.

Sparse encoding of information also appears to be an essential component of associative memory, as demonstrated by the biologically motivated Sparse Associative Memory algorithm (Hoffmann, 2019). In this model, active input neurons are projected onto a pattern-specific sparse set of active neurons, representing the associative memory. This model randomly connects hidden layers to neurons in the input layer and claims that any fixed number of connections to the input layer has no biological plausibility.

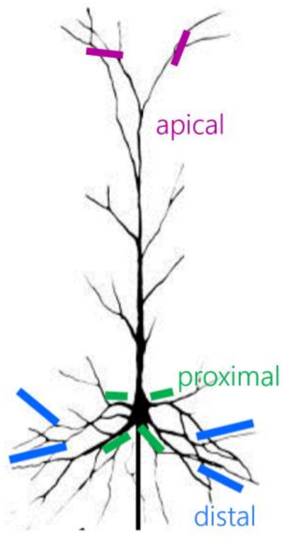
Choosing the optimal Sparse encoding algorithm is an important part of this work. As described later, sparse encoding in this work is used to encode the sensory information into SDR and store the spatially learned patterns and sequences.

### 3.1.2 The model of the neuron

As mentioned in 3.1, biological dendritic excitation suggests a more complex neuron model than typically used in artificial neural networks. The HTM uses the neuron model (Hawkins, Ahmad, 2016) that distinguishes three dendrite segments basal, distal and apical (see 2.5). As shown coloured in Figure 10 A, dendrite segments are classified by the distance from the cells' soma (body of the cell). The nearer segment to the cell, the higher impact on the excitation of the cell. Proximal segments are used in this work to connect to the sensory input and learn spatial patterns. Learning spatial patterns is described in more detail in the next section. Distal (basal) segments are used for learning contextual information. As described in section 5.8, Apical segments are used primarily to build associative connections between areas that host different contextual information.

Neurons that send the information (pre-synaptic neurons) connect with their axons to the receiver (postsynaptic) neurons and form receptor synapses (Figure 10 B). The formed synaptic connection has a strength defined by equation (3). If there is a synaptic connection between two neurons, the synapse that connects neurons is called in HTM, the potential synapse. Biologically, this corresponds to axons that pass close enough to a dendrite segment, forming a (potential) synaptic connection (Figure 10 B).
Every time the pre-synaptic neuron is activated (firing), the synaptic permanence value $p$ is incremented. The increment value *Permanence Increment* used in this work was typically 0.1 or 0.15. In contrast, every time the receiving neuron spikes and its pre-synaptic neuron is not spiking, the permanence value is decremented by the value *PermanenceDecrement*.

A



B

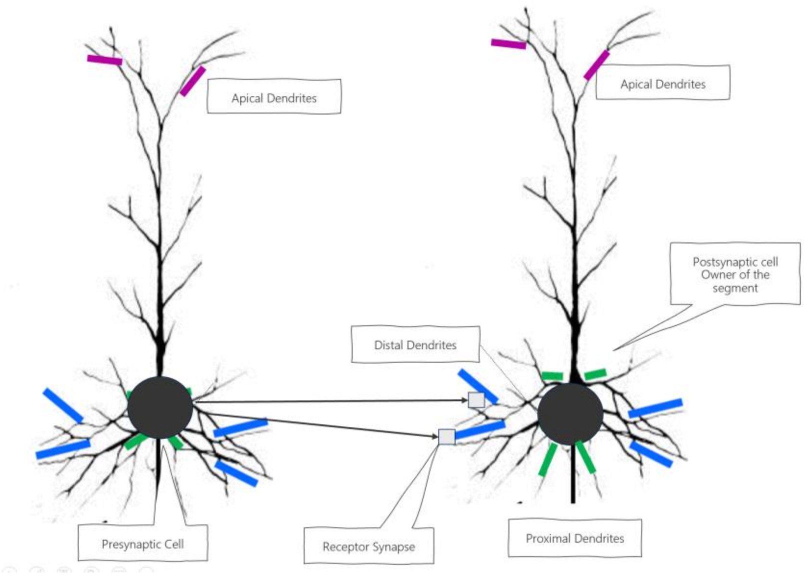


*Figure 10*

*HTM model of the neuron. A) The neuron has three segments: apical, distal (basal) and proximal dendrite. B) A Presynaptic (sending) cell connects with axons to the distal dendrites of the pos-synaptic (receiving) and forms receptor synapses.*

The decrement value used in this work was typically between 0.15 and 0.25. If the decrement value is higher than the increment value, the synapse learns slower than it forgets. Once the permanence of the synapse exceeds the threshold value, *Connected Permanence* $\boldsymbol{\theta_p}$, the potential synapse becomes a connected synapse. In contrast, if permanence falls under the threshold, it will become the potential one. These essential parameters define synaptic plasticity in HTM and directly impact learning.

Another important characteristic of the HTM neuron model is the definition of two states. The first state is the active neuron state (firing state). The second neuron state is the predictive state. The neuron is in the predictive state if the total count of synaptic connections to the basal dendrite segment exceeds a predefined threshold value. As described in section 3.1, these synaptic connections define the context and can activate a neuron without any direct input received at the proximal dendrite segment.

### 3.1.3 Spatial Pooler

The Spatial Pooler is an algorithm used inside HTM for learning spatial patterns (Cui, Ahmad, Hawkins, 2018). The SP receives the spatial input inside layer IV (see 2.4) and encodes it to SDR. Mathematically the SP acts as a function that maps the spatial input $\boldsymbol{I}$ into the sparse code represented as the set of active mini-columns $C^a$ .

$$sp\colon \boldsymbol{I} \rightarrow C^a \qquad (9)$$

The Spatial Pooler forms receptive fields for all mini-columns, as described in sections 2.3 and 3.1. The purpose of the SP is to encode the spatial input into the sparse representation of mini-column states, as shown in Figure 11.

The input of the Spatial Pooler is the SDR typically feedforwarded from sensory cells or cells from other HTM regions described in 2.4. In the case of the sensory input, mini-columns form synaptic connections via proximal dendrite segments (see 2.5) to input sensory cells. The SP learns patterns by following Hebbian learning rules (see 3.1) that continuously strengthen synapses on the repeated appearance of the same pattern (Hawkins, Ahmad, 2016). More details about the mathematical formalisation of HTM CLA can be found in (Mnatzaganian, Fokoué, Kudithipudi, 2016). Please note that SP does not implement the structural plasticity mechanism with rewiring synaptic connections.

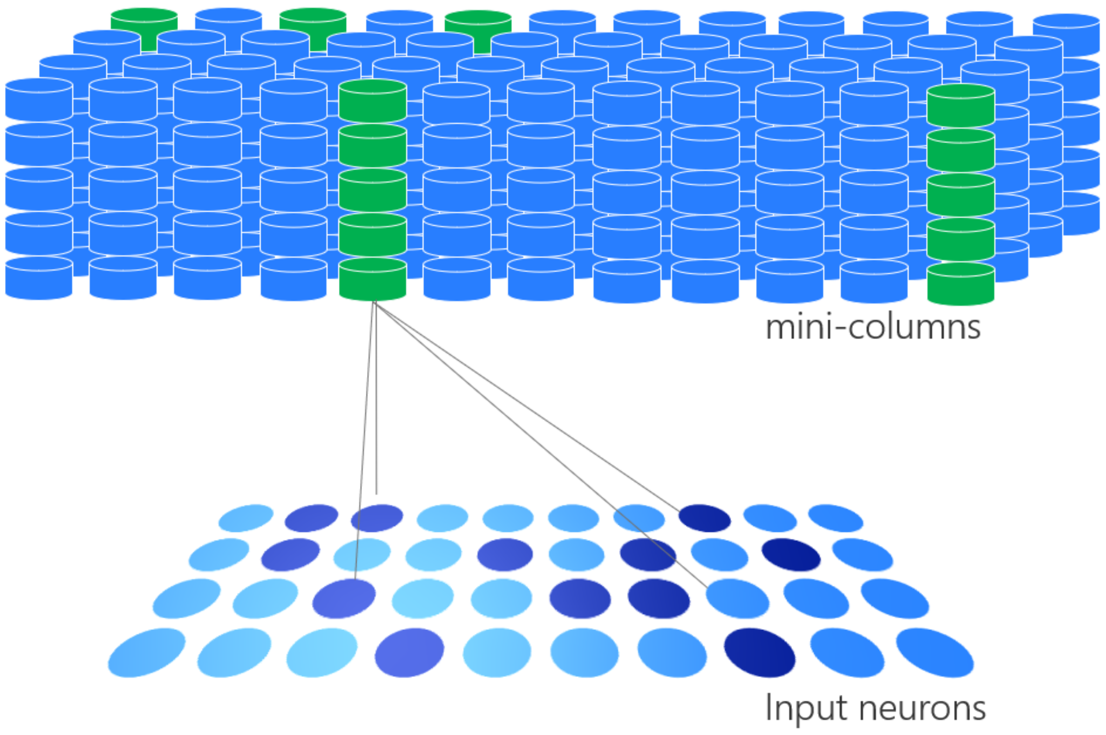


*Figure 11*

*Spatial Pooler builds the receptive field for every mini-column. It connects the neural input cells to mini-columns.*

SP integrates homeostatic excitability control (see 2.6), ensuring neural stability. One of the properties of SP is to produce the same or similar SDR code for similar spatial inputs. It exclusively uses two inhibition (see 2.1) mechanisms: Global- and Local-Inhibition. The Global-Inhibition uses the receptive field (see 2.3 and 3.1) over the entire region. In contrast, the Local-Inhibition uses a smaller area defined by *inhibition radius* and a more complex algorithm, which makes the Local-Inhibition slower than global one. The learning process of the SP is fast and usually takes no more than two or three cycles (iterations) to converge to the input's final SDR code.

The SP is designed as an independent component that learns presented spatial patterns. It uses neurons as input and activates mini-columns without involving mini-column cells in the calculation. The whole mini-column with all its cells is considered active when the pattern is recognised. Section 5.7. provides more details about the design and implementation of the Spatial Pooler used in this work. Moreover, the SP provides noise resistance (Wielgosz, Pietron, Wiatr, 2016), a cortical feature described in chapter 7.

### 3.1.4 Temporal Memory

Temporal memory is the algorithm in the HTM that is responsible for sequence learning. The algorithm, by definition, learns the transition from the cell state $\boldsymbol{c}$ in the previous step $t$-$1$ to the cell state in the current step $t$.

$$tm\colon \boldsymbol{c}^{t-1} \rightarrow \boldsymbol{c}^{t} \qquad (10)$$

The algorithm creates the temporal dependency between the two steps by building the reliance on the previous step. With this, the cell state in the last step defines the context. The input of the TM algorithm is always the SDR (bit-array), which can be the output of the SP, the output of some other TM or similar. The challenge of the HTM modelling is to build the proper flow of SDRs between algorithms.

In contrast to SP, which outputs the SDR of active mini-columns, the TM algorithm acts as a function that maps the SDR into another SDR within the same area.

As suggested in the paper (Hawkins, Ahmad, 2016), learning of sequences biologically might be a part of layer 2/3 (see 2.4, Figure 12 A). The input of the TM is, by default, the set of active mini-columns that recognise some spatial pattern (see the previous section). According to the HTM theory, the spatial pattern is recognised and encoded by the set of active mini-columns. For example, patterns A, B, C, D and X are encoded as unique active mini-column sets. Figure 12 B shows the state of cells before temporal learning but after spatial learning. All patterns are learned, and all cells inside active mini-columns activate the column, but not the particular cell inside the column.

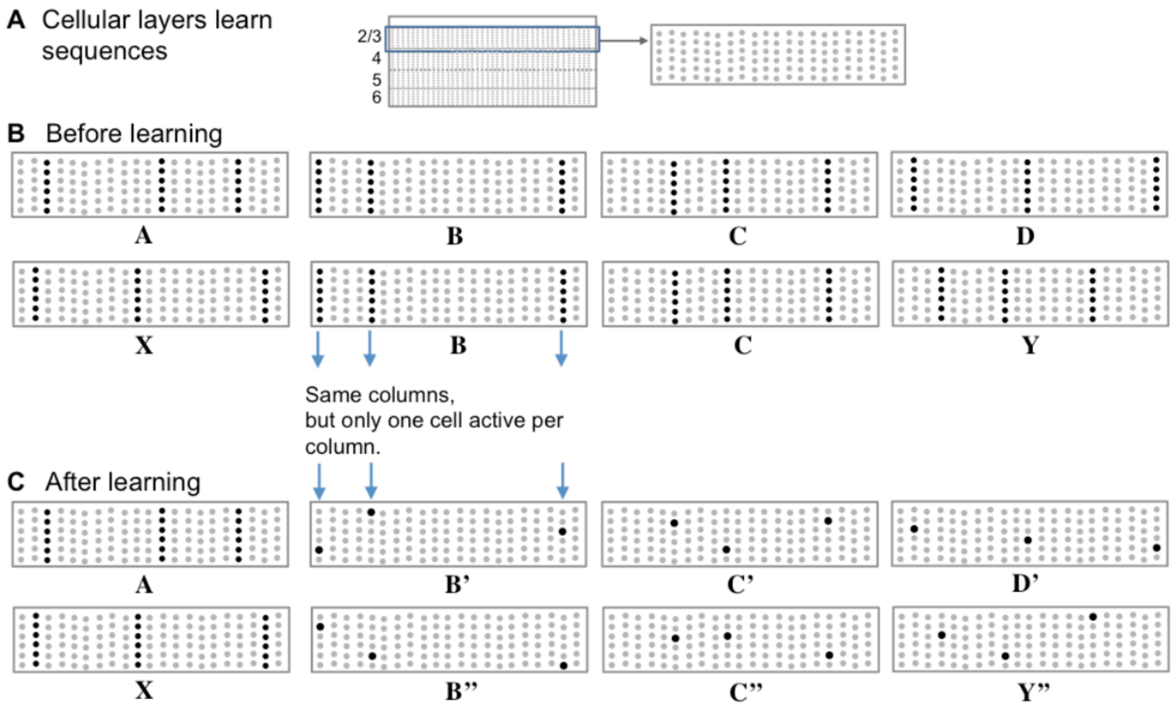

A Cellular layers learn sequences
2/3
4
5
6
B Before learning
A
B
C
D
X
B
C
Y
Same columns, but only one cell active per column.
C After learning
A
B'
C'
D'
X
B''
C''
Y''

*Figure 12*

*Sequence Learning with HTM. Different elements in the sequence (A, B, C or D) are encoded as a unique set of active cells, depending on the context defined by the previous element (Hawkins, Ahmad, 2016). For example, C followed A has a different SDR code than C followed B. However, in both cases, active cells that encode C belong to the same set of active mini-columns.*

After learning the sequence B followed by A, C, followed by B and D followed by C, (A->B->C->D), the TM has activated particular cells inside of the mini-column that represent the spatial pattern (see Figure 12 C). However, when B appears after X, the representation of that B (noted as B′′) is different than the representation of B followed after A (noted as B′). This is because B′ and B′′ share the same active mini-column set but use different cell states. Also, C′ and C′′ that follow B′ and B′′, respectively, are encoded differently.

One of the essential features of the TM is activity-dependent structural plasticity that supports biology findings described in 2.6. With increased activity, the TM can create new segments and synapses and destroy inactive synapses. Section 5.8 provide more details about the structural plasticity model used in this work.

## 3.2 Other related models

In this research, a few more models have been investigated that narrow biological findings to model the cortical algorithm.

### Granger's Model

Granger's model is a model which defines a more specific mapping onto the thalamocortical circuitry (Rodriguez, Whitson, Granger, 2004). The idea behind this model is that there are multiple waves of processing of sensory data. Every wave is differentiated from the previous ones by producing a temporally extended sequence of elaborated categorical encodings.

Synapses in this model are activated and potentiated according to physiologically-based rules. Features incorporated into the model include differential time courses of excitatory vs inhibitory postsynaptic potentials. The model also includes differential axonal arborizations of pyramidal cells vs interneurons and different laminar afferent and projection patterns.

Most aspects of this model, like pyramidal cells and excitatory and inhibitory mechanisms, are widely integrated into HTM.

### The GLOM Model

The basic idea behind this model (Hinton, 1981) is that the spatial and object pathways must learn side by side to generate predictions about what will happen next. Over time this idea of the part-whole hierarchy representation was improved as a GLOM system (Hinton, 2021). The GLOM model suggests how to encode an object as a set of parts (primitive shapes). This work is motivated by the capability of the brain to connect objects semantically to a hierarchy that persists complex objects consisting of simpler objects and shapes. For example, the car consists of wheels, doors, windows etc. The model proposes using columns at five levels. Lower levels in the column recognise simple shapes. The higher the level in the column, the more complex the object is. Elements recognised at lower levels are synoptically connected to objects recognised at higher levels.

Compared to GLOM model, HTM CLA appears more naturally aligned to the biology of the brain. Furthermore, in contrast to Hinton's model, the HTM CLA distinguishes between spatial recognition and temporal prediction calculus as done in the neocortex. The HTM is not only restricted to image procession and provides a more general theory.

### Deep Leabra - a Comprehensive Model of Three Visual Streams

This model (O'Reilly, Wyatte, Rohrlich, 2017) is similar to one of the older version of HTM CLA, which was initially based on the Bayesian generative model. It integrates many biological findings and supports learning with a sophisticated neuron model similar to HTM. The Deep Leabra model focuses processing of visual streams. It tries to demystify the higher-level knowledge built from perceptual experience. It encompasses most of the posterior visual neocortex, the dorsal and ventral. It incorporates two processing streams *What* (the dorsal one) and *Where* (the ventral one). The third information stream connects two named streams. The model records a sequence of video frames where one out of 100 different objects moves and makes random saccades every 200 msec. It assumes that the currently favoured Hebb learning cannot solve real-world problems and that some error signal, like in Backpropagation Algorithm, is required. Deep Leabra shows promising results in a range of applications, including speech recognition, visual object recognition, and text classification. However, the model is still in the early stages of development. More research is needed to evaluate its performance on more complex tasks and optimize its architecture and parameters.

In the meantime, the HTML CLA model was improved and provided a more general theoretical framework.

## 3.3 Conclusion

This chapter summarises many findings in the field of Machine Learning that are essential for this research. It concludes that traditional algorithms solve many industrial problems but cannot be declared Cortical Algorithms even if motivated by some brain features. They are dedicated to solving a specific problem and are not adaptable to be used for any other problem.

The GLOM demonstrates hierarchical learning, but the design does not strictly align with findings in neuroscience as the HTM does. In contrast, Deep Leabra uses some aspects of the cortical algorithm. It delivers a solution that well fits biological findings. However, it is focused on the Image Classification problem and does not provide the general framework as HTM. Some core features of Deep Libra, like the error feedback propagation, are already implicitly integrated into the Cortical Algorithm created in this work. However, how the Cortical Algorithm deals with the error propagation differs entirely from a standard backpropagation approach used in Deep Libra and common Machine Learning algorithms. In addition, using Alpha and Gamma cycles in Deep Leabra is a feature that is not explicitly used in the Cortical Algorithm in this work. As described later in section 9.4, dealing with cycles is a part of the future work. Still, explicit usage of alpha and gamma frequencies is not part of the algorithm and belongs to future work (see 9.5.1.6).

# 4 Parallel programming

Parallelism in computer sciences is a technique of writing programs that execute tasks in parallel on the same machine (also referred to as a node) shared across multiple CPU cores or multiple machines (also called *nodes*) and cores in a cluster environment. This is similar to teamwork in organisations when the work needs to be orchestrated across different individuals, teams and even organisations to exceed the capacity of a single persona. The parallel work needs to be organised, orchestrated and synchronised in a distributed manner, which is generally a complex task. Parallel programming faces very similar challenges.

One of the goals of this work was to find the best technique, model or algorithm capable of efficient parallel execution of the HTM CLA. Several parallelisation approaches, especially in the context of Machine Learning (Pethick, Liddle, Werstein, Huang, 2003), have been analysed to find the best approach. One of the promising approaches is a parallelization strategy based on the message-passing interface (MPI), which allows multiple nodes in a cluster computer to communicate and coordinate their computations.
The original HTM algorithm must be investigated and redesigned to support the execution on multiple CPU cores and nodes. The challenge of this task is a large number of mini-columns and their cells in HTM layers, as described in chapter 2.

## 4.1 The problem with concurrent access to shared resources

The compiled code running in the OS is typically distributed through multiple processes. Depending on the implementation, one process can internally run on multiple threads.
Using multiple threads inside multiple processes is an efficient technique to speed up the execution of the code. This approach can also be extended to numerous machines. In that case, we talk about the distributed environment and parallel execution. Running the code distributed across various machines is in the industry called horizontal scalability.

However, the implementation gets very hard when it comes to concurrent access to shared resources inside a single process, at a single machine and across multiple machines in a distributed environment. Concurrent access introduces a requirement to guarantee mutually exclusive access to the resource. Depending on the application, the protected resource can be a variable, an object, rows in a database etc. Depending on the sequence and timing of the execution, the system can run in an inconsistent state if the access to resources is not mutually exclusive. This is known as a *race condition* (Wheeler, 2004), (Mitchell, 2005). Another critical issue that can occur in this context is called *deadlock*. A deadlock typically occurs when two threads lock the access to the shared resource and wait on the release of the lock at the same time.

The common approach to solving the synchronisation problem is to use resource synchronisation (*locks)* around method execution.
This programming technique is not easy to implement and typically must target a solution in three domains. First is concurrent access from multiple threads to the shared memory inside the same process. This problem is widely solved in modern programming languages using Monitors (Moon, Chang, 2006) (Yang, Kent, Aubanel, Taylor, 2015). Modern programming languages solve this very efficiently by providing language constructs that synchronise code blocks (Wagner, Anderson, Kulikov, 2022).

Second is the concurrent access to resources shared between multiple processes on the same machine (Andrews, 1998). This problem is commonly solved by using mutexes at the level of the operative system (Wilson, 2007) (Microsoft, 2020). This approach allows the code in one process to lock the shared resource. While holding a *lock,* the Code in another method that tries to access the resource will wait until the resource is released.

Finally, the third approach, *distributed lock,* solves the problem of concurrent access to shared resources in a distributed environment (Jonathan Lejeune, 2015). This is a complex problem solved by using many different methods. One of the approaches is a Chubby Lock service (Mike, 2006) developed by Google. The project Apache Zookeeper (Foundation, 2016) is a similar open-source implementation of the same approach. The problem of distributed lock is generally solved but has a drawback. With the increasing number of concurrent access participants, the system's performance does not scale linearly. To be able to parallelise an algorithm, the designer of the distributed system

must redesign the algorithm to be able to partition workloads. With the partitioning of the workload, the concurrent access can be categorised into groups of participants that concurrently access different shared resources.
This approach is one of the significant features of the Actor Programming Model described in the next section.

## 4.2 Actor Programming Model

Object-oriented programming (OOP) is a widely accepted and proven, and familiar programming model. It is a modern programming technique taught by many Computer Sciences education institutions. One of the most fundamental OOP concepts is the concept of "object" and "class" (Xinogalos, 2015).
One of the core pillars of this model is inheritance and encapsulation (Snyder, 1986), which support extensibility and ensure that the private part of an object is invisible outside of the object's context. The object must, by design, expose operations that protect the invariant nature of its encapsulated data.

However, instructions inside invoked methods, executed in parallel on multiple threads or machines, can be interleaved, leading to side-effect called race conditions (see 4.1). Consequently, invariants will probably not remain intact if threads accessing them are not coordinated. As mentioned in the previous section, the problem of coordinating concurrent access to shared resources is a complex task. With a high number of concurrent threads and especially threads shared across multiple physical machines (in this work referred to as *nodes*), implementing such a solution with OO programming languages is complex, expensive and error-prone. Moreover, because the typical OO-based applications usually rely on stateless services, use a relational database as a backend and are modelled by so-called three-tier or even multi-tier architecture, they show weak performance with an increasing number of nodes in the system. Weak performance in this context means that the performance does not increase linearly with the growing number of nodes. Even worse, a growing number of nodes can sometimes lead to stagnation or a decrease in performance.

In contrast, the Actor Programming Model (Hewitt, Bishop, Steiger, 1973) is a mathematical theory and computation model which targets massive concurrency. The Actor model builds on existing models of nondeterministic computation, for example, Turing's model, Petri nets etc., and sets the focus on concurrency. In this theory, the

“Actor” is treated as the universal primitives of concurrent computation (Hewitt, 2010). An Actor is a computational unit (see Figure 13) that can concurrently run code in response to an asynchronous message it receives. By default, the actor algorithm inside common frameworks typically processes incoming messages unordered. However, if an application requires ordered message processing, a single batch-message can be serialized as a sequence of messages processed in the order.
The motivation for this programming model in this work is more simple reasoning about concurrent computation. It allows one to think about code in terms of communication. This reasoning is based on the fact that computation units, which execute concurrently, are contextually independent and can be executed in parallel without the need to provide a distributed lock mechanism, which always limits performance at a large scale. The Actor Programming Model (APM) introduces the concept that does not require concurrent access control in an explicit way as OO does. As shown in Figure 13, the actor executes in a single thread and holds its state (resources) persisted to some volatile or durable store. Messages are received and stored in the queue before being processed. The implementation of the APM framework internally dispatches messages to the destination actor, whose instance is identified by the actor type and actor ID. With this concept, two messages cannot trigger the concurrent execution of the same code in the same context, defined by the particular actor instance. It appears, at first, that the actor cannot increase performance because it cannot scale out of a single thread. Adding more CPU cores to the machine that executes the same actor instance would not increase the performance. APM can only support efficient parallel execution if the computation can be partitioned into many actor instances.  In that case, multiple actor instances can run concurrently without synchronization. This very simple approach makes it possible to quickly implement the highly distributed code by running many actors. All resources that need to be concurrently shared are allocated only to actors that need to access them. An actor has no constructor as a class in the OOP model.  Theoretically, an actor virtually always exists. It is executed when it is needed and released when it becomes inactive. By definition, the code sending the message to the actor has no knowledge about the cluster where actors will be executed. This message routing detail is called *location transparency*. It is transparent to the developer and simplifies the implementation of the actor itself and the code that consumes the actor’s functionality.

Implementing the APM must be theoretically split into two physically separate parts. The first part will be where the user's code is physically executed. The second part is the computation at the remote node in an actor.
These two parts play the role of scatter and gather (Hearst, Karger, Pedersen, 1996). Such an approach describes a way of addressing the actor. The scatter code prepares all required parameters for executing the code inside an actor. It acts as a proxy between the user's code and the executing code in the actor. The gather is the opposite side inside the framework. It receives the parameters from the scatter and dispatches the execution to the actor.

Due to the increasing popularity of cloud technologies, more and more applications are becoming distributed and require programming models that make this kind of computing easier (Dobric, 2016). However, another issue today might be a driving factor for the Actor Programming Model. The CPUs are not getting faster than in the last decade. Some programming frameworks (JAVA, .NET, NodeJS etc.) widely accepted in the industry target multicore programming (single physical machine with multiple CPUs) and solve this problem efficiently (Leijen, Schulte, Burckhardt, 2009). Running computational logic across many physical nodes with multiple CPUs represents a significant challenge. This is where the Actor Programming Model is, in some scenarios, a great alternative.

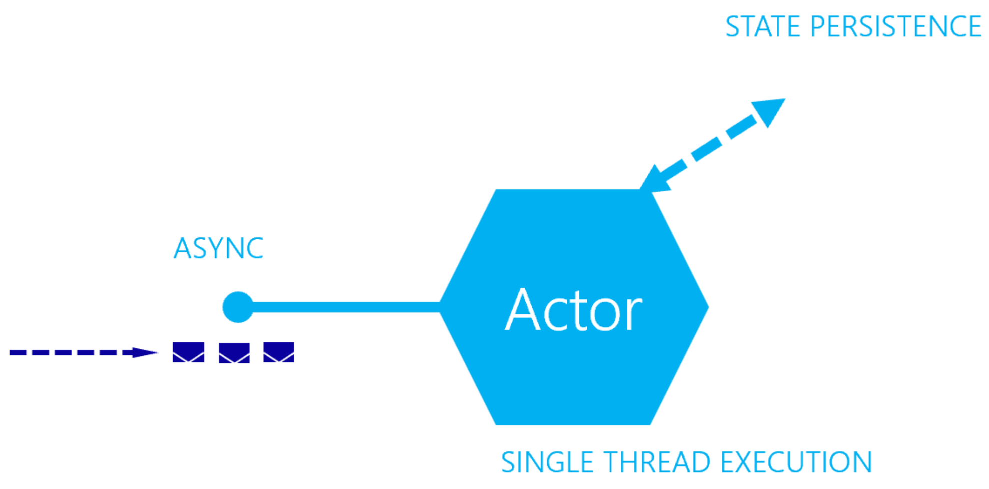


*Figure 13*

*The model of the actor. Messages are received and executed in a single thread. The actor persists internal state to the volatile or durable storage.*

To find out the actor framework that best fits the requirements of this work, several newer and modern Actor Programming Model implementations have been evaluated:

*Orleans Virtual Actors*
The Orleans (Microsoft, 2019) is an open-source Actor Programming Model Implementation mainly used internally by Microsoft and Microsoft partners. It was initially designed by Microsoft Research (Bernstein, Bykov, Geller, Kliot, Thelin, 2014). This model was originally implemented as a backend for the game “Hello” and found limited adoption in the community and several enterprise projects.

*Service Fabric Reliable Actors*
The Service Fabric is the container technology that supports the Actor Programming Model (Microsoft Corporation, 2016). It is similar to the *Orleans* Virtual Actors currently used as a backend of many services in the Microsoft Azure platform (Dobric, 2016). The drawback of this model is a strong dependency on the Service Fabric cluster technology. Because this product will be consequently merged with the Kubernetes technology, the actor model framework will be discontinued. For this reason, this framework will no longer be followed in this work.

*Akka.NET*
The open-source implementation of the Actor Programming Model (Actors, 2015) (Petabridge, 2016) is widely used for enterprise software development. JAVA and .NET developer communities adopt this framework widely because it allows OO developers to deal easily with the APM approach. However, the drawback of this model is the complexity of the maintenance of the cluster and the limited configuration of native internal execution details.

*Azure Durable Functions*
The stateful serverless programming model supports durable entities on top of Actor Model principles (Microsoft, 2021). This framework represents a trendy

Serverless Stateful Programming Model. Developers are not required to manage a cluster of nodes because the serverless platform serves and automatically maintains the computation nodes.

As described later in chapter 6, many ideas from the listed frameworks were integrated with this work. However, due to the specific requirements of the HTM model, it was not possible to fully meet these requirements. For example, the actors' focus within *Orleans* is to provide a framework for high-scaled computation of a short computation logic implemented in modules called grains. The framework does not provide a control of the instantiation of *grains* from the client side. This is an advantage for developers because they do not need to care about the destination of the code in the *grain* (actor). However, this design enforces the framework to persist the state of the actor right after the execution or to provide communication between nodes to maintain the life cycle of the actor. Both the persisting of the state and the internal communication slow down the performance, especially when the state occupies a large amount of memory, as in the case of Cortical Learning Algorithms. Depending on the implementation, persisting of the state of an actor usually might take more than ten to a hundred times longer than the execution. The same limitation was found in the Akka.Net framework—the running of the HTM-CLA in these two frameworks led to high-CPU, craching and overload of the cluster.

Finally, the Azure Durable Functions framework enables state maintenance and code execution with the Actor Model principals. However, the drawback of this model is that it does not support any configuration and control of message routing and internal code execution. Moreover, it is designed for typical enterprise applications. It is robust, reliable and easy to use but generally too slow and expensive to serve many actors representing cortical mini-columns.

As a result of testing all named frameworks, it was decided that all of them were not suitable for implementation of the CLA. For this reason, a new lightweight Actor Model framework was designed and implemented (see 6.2).

# 5 Modelling of a Cortical Learning Algorithm

Following different computation intelligence approaches discussed in chapter 3, the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) was chosen as the best match to the biological findings described in chapter 2. In contrast, all other traditional (standard) algorithms are well-designed to solve problems in the specific problem domain. Still, they can not be a foundation for building a unified cortical algorithm.

The main objective of this work was to create the Cortical Learning Algorithm by following the principles of the HTM CLA. This chapter describes the Algorithm designed and implemented in this work. It starts with the memorizing capacity (5.1), followed by the model of computational plasticity (5.2). Next, the model of the artificial cortical area and associated receptive field are described in section 5.3. Finally, sections 5.4 and 5.5 describe synaptic and segment activation mechanisms and encoding model, respectively.

Sections 5.7 and 5.9 describe how required parts of HTML CLA are integrated into the algorithm created in this work called the *Neural Association Algorithm* (NAA).

Section 5.8 describes the NAA that extends the HTM CLA and provides a more general Cortical Learning Algorithm. This algorithm is a theoretical framework that fully implements the part related to HTM CLA. Some other parts of this algorithm are still in development. The implementation in .NET C# is published as an open-source framework called *neocortexapi* (Dobric, 2019). Implementing the HTM part of the NAA is strictly aligned with the original Java implementation, even though this approach does not fit naturally to the .NET Core and C# language. This decision was made to compare scientific results across different frameworks better.

## 5.1 Huge memory capacity in the small space

It is commonly believed that computer memory is very limited compared to the brain's memory. This belief derives from conventional thinking related to the standard computation technology and memory limitations. However, the sparse neural information persistence described in 3.1.1 suggests an entirely different approach that enables a vast capacity on commodity hardware. To get an idea of memorizing capacity of the neuron population, assume the number of neurons is just $n=16$, and every pattern is represented by $a=3$ active neurons (see Figure 14). Then, applying equation (5) with these parameters, the memory in the *neocortexapi* can save 560 bits in only 16 available physical bits.

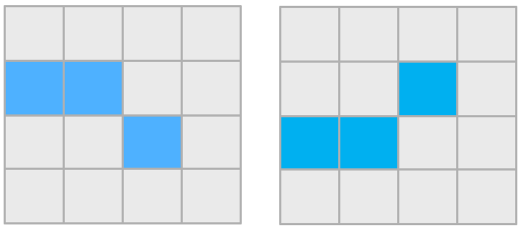


*Figure 14*

*Sparse encoding of the pattern. Two different patterns encoded with 3 bits in the space of 16 bits, which define sparse factor (sparsity) 3/16=0.187. The maximum number of stored patterns is 560.*

By using the same equation, together with the sparsity of approx. 2% of neuron population set as described in 3.1.1, the memory capacity of the population of neurons is calculated for values n from $n=500$ to n=$4096$ bits. These values were chosen because they are reasonably small numbers that can be used in experiments on commodity hardware. They are also large enough to recognize patterns with described sparsity. The capacity as a function of the number of neural cells by a fixed density of 2% is shown in Figure 15. For example, the population of 2048 neurons with 40 active neurons (sparsity 2%), gives the capacity of $10^{84}$ possible patterns.

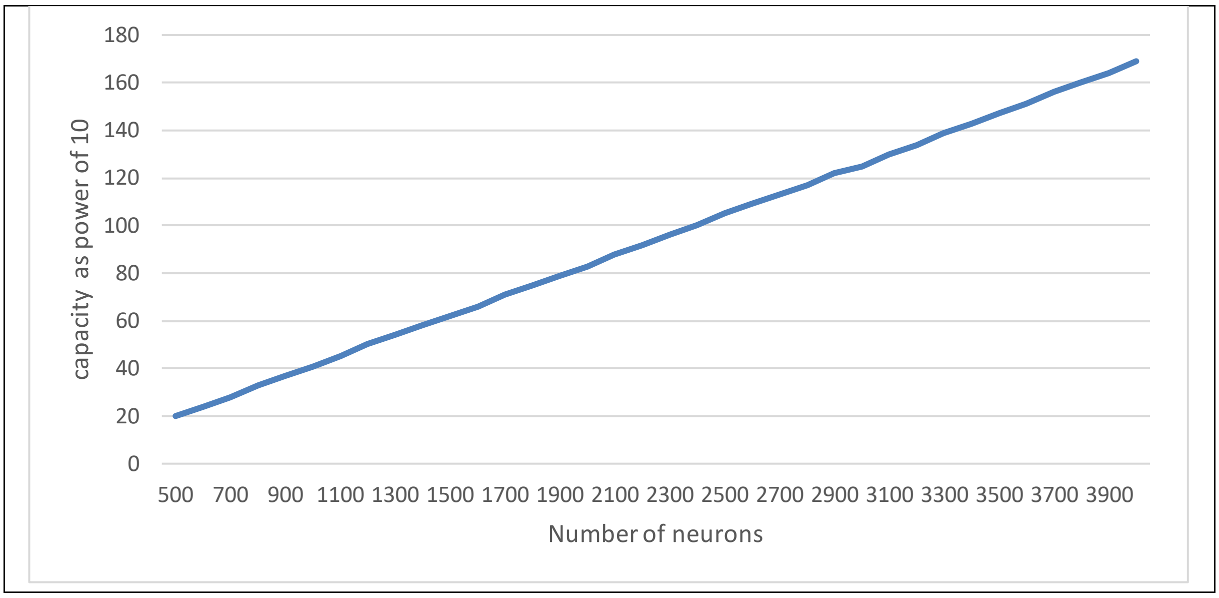


*Figure 15*

*The number of patterns represented as a power of 10 that the SDR can memorize as a function of the number of neural cells in SDR with 2% active neurons. The vertical axis represents a capacity as a number of patterns that can be stored. Horizontal axes represent the number of available neurons in the population formed by HTM.*

According to the results shown in the previous diagram, it can be concluded that even a minimal number of neurons (i.e. 2048 or 4096) can store a huge number of patterns. Following the biological functioning of synapses described in section 2.1 and equation (6), the HTM recognizes the pattern if the number of active neurons, synapses or mini-columns is greater than the activation threshold $\boldsymbol{\theta}$. For example, if 40 neurons are defined to be active by the encoding of some pattern (2% of 2000 mini-columns) and the $\boldsymbol{\theta} = \mathbf{30}$, then the pattern is defined as recognized if 30 or more neurons are active. Theoretically, the threshold $\boldsymbol{\theta}$ can be set to the number of available bits in the population (in this example $\boldsymbol{\theta} = \mathbf{40}$). The implementation of *neocortexapi* can handle this naturally. However, the algorithm might sometimes predict ambiguous patterns with insufficient contextual information. However, for practical reasons, according to 2.6 and 3.1.1, some other value $\boldsymbol{\theta} < \boldsymbol{n}$ should be chosen. There is no precise threshold that can be used to deliver the best result, but experiments in this work suggest using a threshold $\boldsymbol{\theta} \approx 0.7\boldsymbol{n}$.

Because every neuron in the population recognizes multiple patterns (see 2.6), the probability of false positives must also be considered. By following equations (7) and

(8), the false-positive error probability of the sparse representation in HTM can be calculated. For example, Table 1 shows error probabilities for *a=40* and *n=600*.

| **Threshold** | **Error probability** |
|---|---|
| 40 | 2.307E-63 |
| 35 | 6.842E-46 |
| 30 | 1.507E-33 |
| 20 | 8.530E-16 |

*Table 1*
*List of error probabilities for activation thresholds of 40, 35, 30 and 20 by SDR with 600 neurons(n=600) and 40 active neurons (a=40).*

Even by the threshold of 20 recognized neurons, which is 50% of 40 neurons used for encoding in the population of 600 neurons by the sparsity of 0.0667 (40/600), the error probability is extremely low, just 8.530E-16. By *a=40*, there are 40 synapses forming connections to active neurons. Therefore, with an expected threshold of 20, the recognized synapses neuron will generate the NMDA spike (see 2.6) with an error probability of 8.530E-16. To be sure that using suggested populations of neurons in this project in the range between approximately 500 and 4096 is feasible, the error probabilities of false positives have been calculated. For this calculation, the sparsity of 2% was used, which. gives a different number of active neurons for every defined population. Also, the threshold $\boldsymbol{Q}$ was used in the range $\boldsymbol{a} > \boldsymbol{\theta} > \mathbf{0.3}\boldsymbol{a}$. The result is shown in Figure 16.

The result in the figure implies the following findings. First, the error probability of the sparse representation is extremely low. Second, the higher number of neurons in the population leads to a lower error probability.

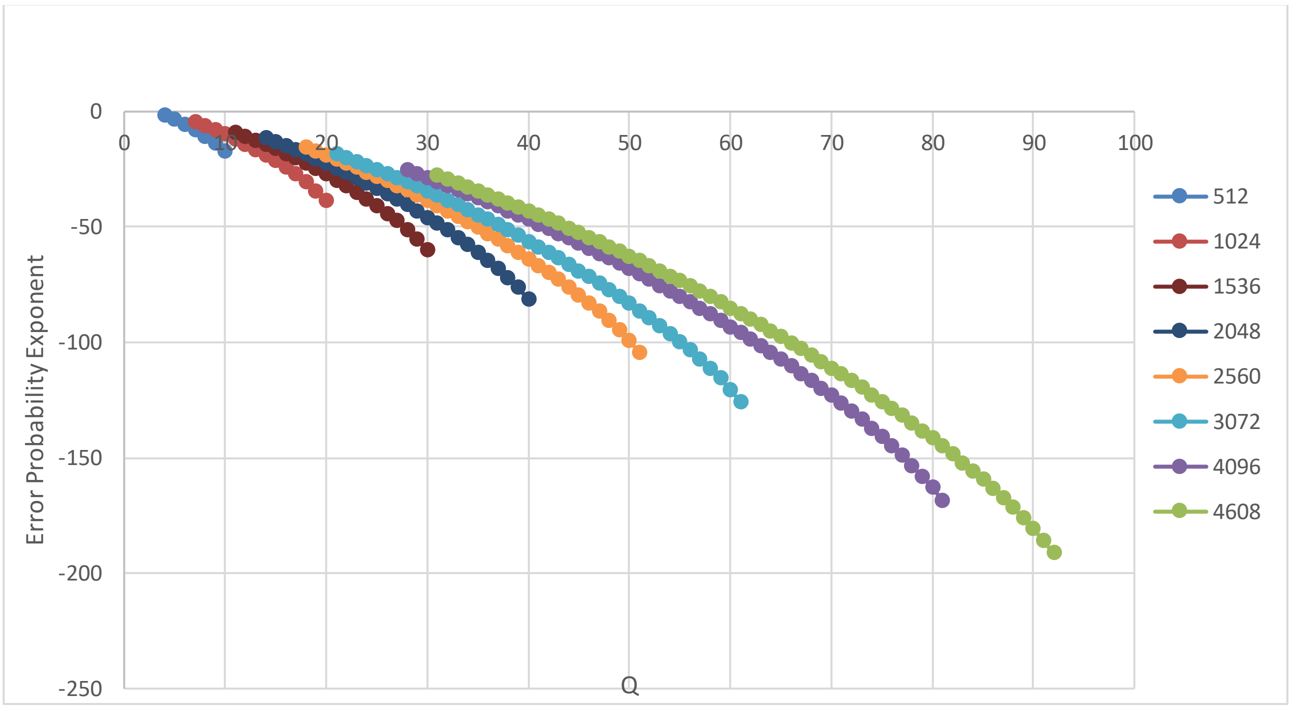


*Figure 16*

*Error probability of the false positives. The result is shown for different populations of neurons mainly used in experiments in this work. The X-axis represents the threshold Q, which is the number of neurons required to recognize the pattern. The Y-axis shows the exponent value of error probability of the population by a given threshold Q.*

That means increasing the number of neurons in the artificial tissue model can recognize patterns with decreasing error probability. Third, the error probability of the population with fewer neurons can be reduced by choosing the higher threshold. Fourth, threshold values with even just 50% of active neurons of the population still generate an extremely low error probability.

## 5.2 Synaptic plasticity and the Overlap function

The idea behind the HTM algorithm relies rather on the cell population's activity and instead on a single neuron's activity. Following the encoding mechanism discussed in 3.1, the HTM in this implementation (Dobric, 2019) also models neuron activations and synapse weights with binary states. According to neocortical theory, they align with biological synapses, have stochastic nature (see 2.6), and do not require synaptic precision. The synaptic connection is considered if its *Permanence* (see 3.1.2) exceeds the *Connected Permanence* threshold value $Q_c$. This threshold switches the binary state of the synapse in the HTM. As long the permanence is under this value, the synapse is not-connected, and it is not involved in the internal calculation. Not-connected synapses are called *Potential Synapses*. This value makes the *potential synapse* becomes a *connected synapse* and vice versa. Continuous adoption of the permanence value of the synapse is motivated by synaptic plasticity, described in section 2.1.

The overlap function is one of the most critical functions in the created algorithm. It counts the number of connected synapses (active synaptic connections) of postsynaptic neurons to pre-synaptic active neurons. For example, assume that two populations of neurons, P1 and P2, are synaptically connected. The population P1 creates presynaptic connections to dendrites of neurons of the population P2. From the point of view of P2, the overlap function is defined as the number of connected synapses to active (firing) neurons of the population P1. Synapses connected to inactive (non-firing) neurons in the population P1 are not counted in the overlap because they are binary switched off. If the $c_j^2$ is the binary representation of the potential connection from some cell of the population P2 to a cell $c_i^1$ of the population P1, then the overlap between populations P1 and P2 is calculated as:

$$o = \{\sum c_j^1 * c_i^2\} \equiv \|P_1 \wedge P_2\| \qquad (11)$$

The overlap function calculation is simple to implement and very fast operation. Analog to overlap, the *Percentage Overlap* is a ratio between the number of synapses connected to active neurons and the total number of neurons. For example, if one cell from P2 is connected to 1000 neurons from P1 and 100 of 1000 neurons are active, then the overlap is 100. Analogue, the *percentage overlap* is defined as the ratio is 100/1000 = 0.1 (10%).

The synaptic permanence, potential connection, active and predictive neuron states, and the overlap function described in this section are the fundament for implementing the cortical algorithm in this work.

## 5.3 Modelling the Hierarchical Temporal Memory cortical area

The HTM algorithm created in this work is designed to model a cortical area with almost any topology in order to investigate the resulting cognitive capabilities. The simplified HTM cortical area is shown in Figure 17 left. It consists of mini-columns that cross all six layers inside a hypothetical cortical column previously described in section 2.4. Note that connections between cells inside and between mini-columns are omitted in the figure. However, to create such an area, the implementation must provide some concept of a cortical network consisting of interconnected smaller cognitive units.
The cognitive unit used in this work is defined as a set of mini-columns that belong to the same biological layer described in 2.4. That means the cognitive unit forms the so-called HTM layer that biologically represents a part of a cortical column that fits a single cortical layer. Figure 17 right shows the model of such a cognitive unit, the smallest canonical unit used in this research. In the example presented in the figure, the layer builds connections to the sensory input, which corresponds to the cortical layer IV. The layer in Figure 17 right can form the HTM model shown in Figure 8 with Spatial Pooler, Temporal Memory, or even some other future algorithm.

When modelling multiple cortical layers, like in Figure 9, the input of some layers is the output of the preceding layer(s) or the sensory input (Lewis, Purdy, Ahmad, Hawkins, 2019).

During the layer's initialisation, mini-columns build a *receptive field (RF)* with synaptic connections to their input cells. In the example in Figure 17, bottom right, sensory cells are input for the layer, and every mini-column creates synaptic connections to the subset of sensory cells. Connections are built from presynaptic input cells to proximal dendrite segments (see 3.1). Neighbourhood mini-columns typically overlap their *RF,* as shown in Figure 17. With this, neighbourhood mini-columns build redundant connections to input cells, a biological robustness mechanism.

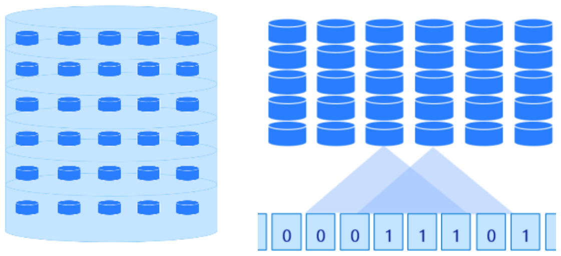


*Figure 17*
*Left: Simplified representation of the mini-columns inside of a* cortical column *of an area across all layers (layers boundaries are omitted). Right: Shows a cortical unit forming the HTM Layer IV overlapping the receptive field of synaptic connections to sensory input.*

The examples in Figure 17 right show a receptive *field* of two mini-columns with synaptic connections to six sensory input cells each. Every mini-column span over the column dimensional space with the uniform *column ratio* set *CS:*

$$CS = \left\{\frac{k}{colNum}; k \in \{0,..,colNum - 1\}\right\} \tag{12}$$

With given four columns (example) column ratio set looks like this:

$$\{0,\frac{1}{4},\frac{2}{4},\frac{3}{4}\} \tag{13}$$

Every element in the *CS* < a subset of the sensory input, potentially connected to the mini-column cell. The topology can be defined in multiple dimensions. Given $id$ and $cd$ as numbers of input and output dimensions, respectively, as well as $I^{\mathrm{id}}$ and $C^{\mathrm{cd}}$ multidimensional input and column spaces, the *column-span* $SI$ *over* the sensory input space $I^{\mathrm{id}}$ in every dimension is defined as:

$$span = \left\{\frac{colDim_i}{inDim_i}; i \in \{1,..,\min(id,cd)\}\right\} \tag{14}$$

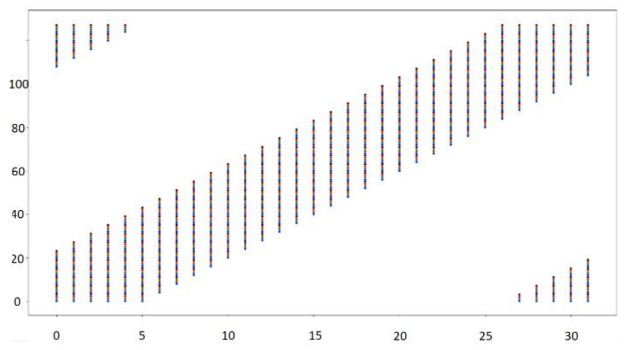
*SI* defines the ratio, which represents how many sensory input cells fit in the receptive field (*RF*) of a single mini-column with respect to uniform *RF* share across all mini-columns. This value, in combination with *CS*, is used to calculate the centre of the *RF* of the mini-column $c_k$, by function $cen(c_k)$. From the centre, every mini-column builds its RF over the area defined by a parameter *potential* radius $r$. Within a possible radius, every mini-column is potentially connected to the input. In generalized form, the *receptive field* RF is a set of input cells $i_j$ With index $j$ connected to a mini-column with the index $k$:

$$RF^{\mathrm{id}} = cen(c_k) - r < i_j < cen(c_k) + r \mid RF^{\mathrm{id}} \subset I^{\mathrm{id}}, i_j \in I^{\mathrm{id}}, c_k \in C^{\mathrm{cd}} \quad (15)$$

Figure 18 A - shows a distribution of the receptive field of the layer with 128 mini-columns and 32 sensory input cells with a radius of 5. The vertical axis is the index of the mini-column, and the horizontal axis is the index of the cell to which the mini-column is potentially connected. It shows that mini-columns overlap their receptive field. Figure 18 B shows a zoomed area indicating the centre of the RF of the mini-column.

In the case of multiple dimensions, both mini-columns and sensory input are flattened. For example, sensory input 32x32 will be represented by an array of 1024 cells, and mini-columns 64x64 will be represented as an array of 4096 cells (see Figure 19).

A



B

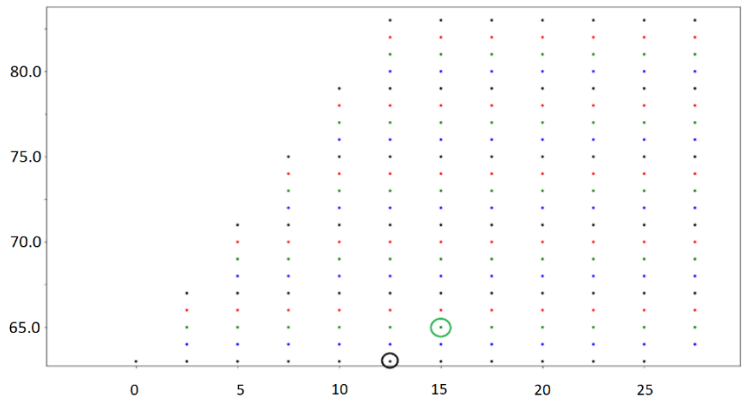


*Figure 18*

*Receptive Field of the HTM layer with 128 mini-columns, sensory input of 32 cells and radius 5. The vertical axis is an index of the mini-column. The horizontal axis is an index of sensory input cells. Note that colours in rows have no other meaning than visually separate rows. A) Shows all RF sets of the layer. Increasing the mini-column index shifts its RF. B) Shows a zoomed snapshot of cells (columns 10-21). For example, mini-column centres are marked from the bottom for the 1st and 3rd rows. Because the radius is 5, left and right from the centre, 5 cells are included in RF.*

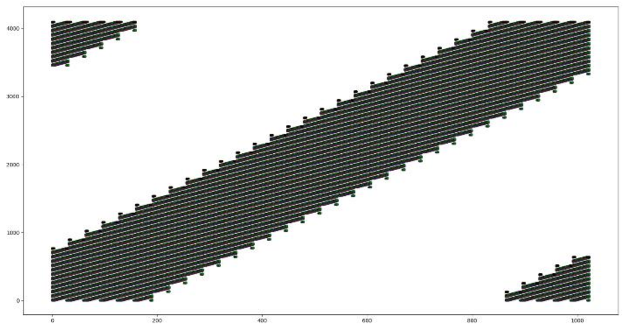


*Figure 19*

*Represents a cortical column's receptive field (RF) with 64x64 mini-columns and 32x32 sensory input cells. Dimensions are flattened as 4096 and 1024 cells for mini-columns and sensory input cells, respectively.*

However, not every input cell within an RF defined by equation (15) will be connected to a mini-column. The percentage value of the RF's subset *Potential Percentage* $RFp$ defines a percentage of input cells within RF, which will be randomly connected to a specific mini-column. Currently, there is no biological evidence for this behaviour in the tissue. However, the HTM implements this as a possible mechanism that can be used in experiments. For example, Figure 20 shows RF with a radius of 5, which according to equation (15), occupies 10 cells. By $RFp = 0.5$ (Figure 20 right) 5 of 10 cells within an

RF will be connected by synapses to mini-columns. Figure 20 left shows the RF with $RFp = 1.0$, where all cells within RF are connected to selected input cells.

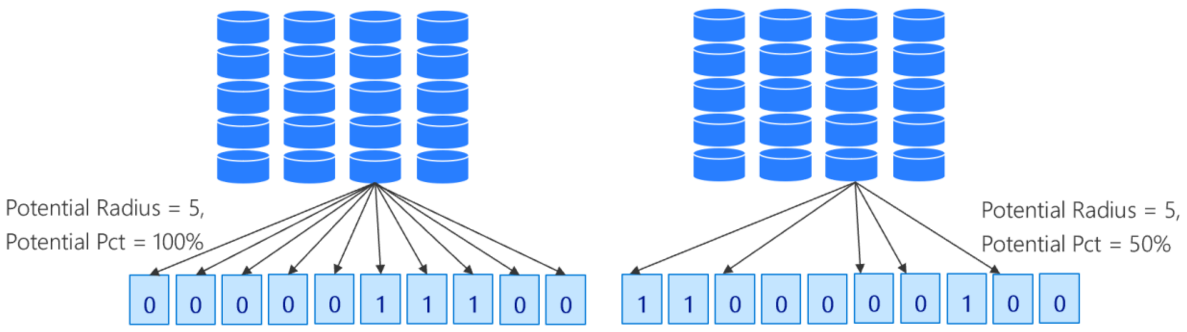


*Figure 20*

*Mini-Column receptive field with potential percentage value 1.0 left and 0.5 right.*

The overlap of the RF between mini-columns in the neighbourhood increases the robustness against damage in neurons. If, for example, one of the input cells would be damaged, many other cells share many of the same connections to mini-columns and vice versa, as shown in Figure 18 B.

## 5.4 Activation of synapses and dendrite segments

As previously described in 2.1, synapses connect the source cell's axons with the receiving cell's dendrites. The HTM model potentially combines a *dendrite segment* ***D***, owner of the synapse, with a neural cell called *source cell* or *presynaptic cell* $\boldsymbol{c_s}$ . During the learning process, every repeating input strengthens the synaptic permanence value. This is a very simple and fast computational operation defined by the Hebbian rule (see 3.1). In contrast to standard neuronal networks, binary synaptic states in conjunction with described Hebb rule is easy to compute and require much fewer computation resources than the backpropagation algorithm.

Figure 21 shows the model of dendrite segments used in this work. Proximal dendrites (green) are directly connected to the input sensors. Distal or basal segments (blue) connect to cells of other mini-columns inside the same area (region) described in the previous section. Apical segments (magenta) build synaptic connections to cells from other areas (regions).

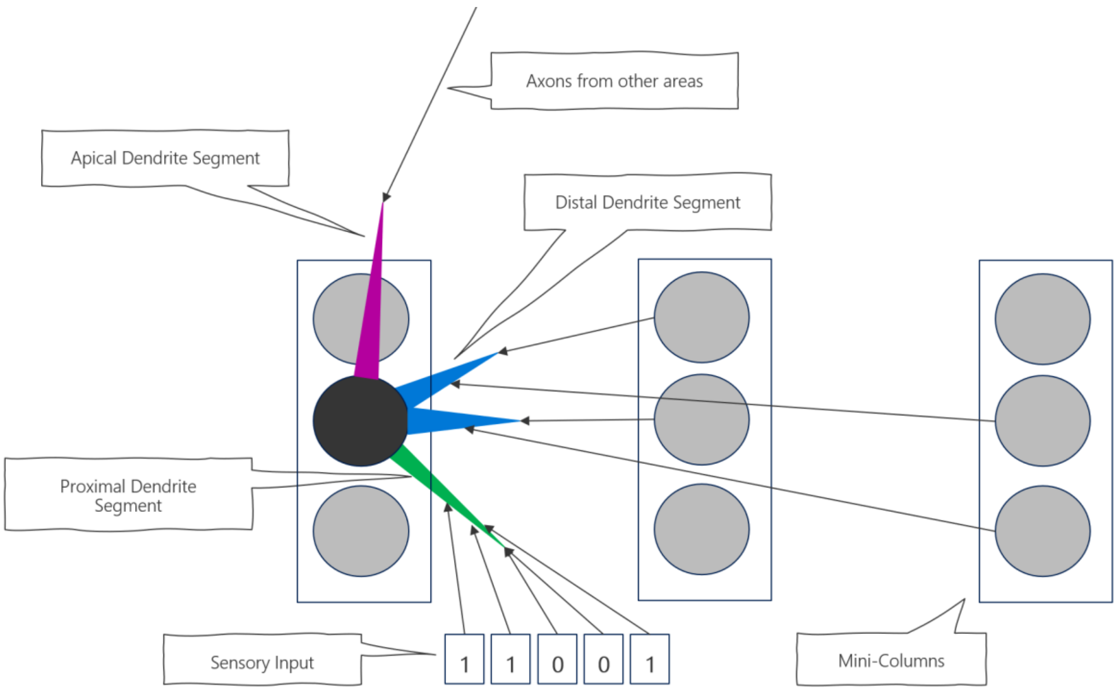


*Figure 21*

*Dendrite Model of the neural cell. Green: Proximal Dendrite Segment connecting to sensory input. Blue: Distal Dendrite Segments connecting to cells from other mini-columns inside the same area (region). Magenta: Apical Dendrite Segments connecting to cells from other areas (regions)*

The model distinguishes between two activation rules. The first one is used to activate (select) the mini-column when the number of connected synapses on the proximal segment (green) exceeds the defined column stimulus threshold $\boldsymbol{\theta_p}$. The second and third ones are used to activate the distal and apical segments when the number of connected synapses exceeds the activation threshold $\boldsymbol{\theta_d}$ and $\boldsymbol{\theta_a}$ respectively.

A synapse is, by default *connected* if its permanence value is greater than a defined synapse *Permanence Connected Threshold* $\boldsymbol{\theta_c}$ (see 5.2). Even if the synapse holds two states, its permanence value represents the postsynaptic cell's activation probability when the presynaptic cell is activated. When a sparsely encoded pattern is recognized, a segment topology creates *n* pre-synaptic potential connections from other cells to segment cells as defined by equation (15). Formally, the whole segment can be represented as a binary vector:

$$\boldsymbol{D} = [b_0, b_1, .., b_{n-1}] \qquad (16)$$

where $b_i$ are non-zero values, and $i$ is an index of presynaptic connection independent of the segment type. With $\boldsymbol{s} = |\boldsymbol{D}|$ is the segment's defined number of potential connections. When the pattern is recognized, the subset of connections $\boldsymbol{D}'$ inside of the segment is activated:

$$\boldsymbol{D}' \subset \boldsymbol{D}; s \ll n \qquad (17)$$

A dendritic segment is considered *active* if it holds more connected synapses (as defined by equation (11) of active cells on that segment than the segment activation threshold $\boldsymbol{\theta_d}$. Finally, the neuron is considered as the active one (firing) if it owns the active segment.

To recap, synaptic permanence value, learned by the Hebb rule, defines whether the synapse is in a connected (active) or potential (inactive) state. The binary state of synapses on the proximal segment activates the mini-column if the total count of connected synapses on the proximal segment is greater than the synapse threshold $\boldsymbol{\theta_p}$. The binary state of synapses on the distal and apical segment activates the segment if the total count of synapses of active cells is greater than the segment activation threshold $\boldsymbol{\theta_d}$ and $\boldsymbol{\theta_a}$ and respectively.

With this, the described HTM model in this work drives the activation of synapses, neural cells and dendrite segments.

## 5.5 Sensory Input Encoding

Section 5.2 describes the HTM using binary states of neurons and synapses to perform internal calculus. For this reason, sensory cells must be capable of transforming the sensory information into the binary code, represented as an SDR. By design, any information represented in this form can be used as input for the cortical algorithm. The HTM provides an encoder concept suggesting how any sensory input can be encoded as an SDR. This work has implemented various encoders like *Scalar Encoder,* *DateTime Encoder*, *Category Encoder, GeoSpatialEncoder, BooleanEncoder* and more. The primary role in this work is a Scalar Encoder. It is used to encode the scalar numbers from a specific range. All other named encoders rely on the concept of the encoding of a scalar number.

The encoder concept described in 2.6 inspires the implementation of encoding of different types of sensory inputs like motion, temperature, sound, pressure or almost anything else. This biological concept encodes similar values with a high number of overlapped bits. Once the data is encoded as SDR, the HTM will process it naturally as it would be sourced by any possible sensory cell.

A scalar Encoder uses several parameters that define the encoding algorithm. The first important parameter is the number of bits *N* in the SDR that will be used as a spatial output of the encoder. The parameter *W* (width) specifies how many non-zero bits will be used to represent a single scalar value. For example, the following vector represents an SDR with the total number of output bits *N*=10 and the width *W*=4 of non-zero bits:

0011110000

The encoder does not encode the input using any known code like ASCII or similar. The output of the encoder rather represents the energy concentration around firing neurons in a biological way. For example, two similar values, 6 (0011110000) and 7 (0000111100), are encoded with more overlapping bits than values 0 (1111000000). and 9 (0000001111). Encoding the same values in ASCII or binary code would lead to losing the biological similarity of nearby values. Technically, the Scalar Encoder always requires the possible encoding range for practical reasons. The values Minimum and Maximum define this range. For example, encoding the voltage in the household would require encoding in a range of 0 – 220 V. This should not be understood as a limitation. It is rather biologically inspired. All sensory cells in an organism are always limited in sensing a range of values. Given the number of bits N and the width of an encoded value, the number of possible uniquely encoded values $E_{min}$ without overlapping of non-zero bits can be calculated as:

$$\mathrm{Emin} = \frac{N}{W} \mid W < N, W >= 1 \tag{18}$$

Because there is no overlapping, this is the minimum number of possible uniquely encoded patterns for given N and W. The maximum encoded value $E_{max}$ can be calculated when shifting the non-zero bits sequence by a single bit:

$$\mathrm{Emax} = \boldsymbol{N} - \boldsymbol{W} + \boldsymbol{1} \mid \boldsymbol{W} < \boldsymbol{N} \tag{19}$$

Figure 22 shows two encoding examples. Example A shows the encoding of 13 possible values with the given number of bits N=15 and width W=3.

```
(A)

1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
```

```
(B)

1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
```

```
(C)

1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
```

*Figure 22*

*Represents three examples of the encoding of a range of scalar values. A) Encoding of 13 possible values for N=15 and W=3. Yellow marked bits overlap by encoding two nearby values. B) Encoding of 9 possible values for N=10 and W=3. C) Inefficient encoding with the same SDR for two different values. Yellow rows show two same SDRs that encode two different values. In this case, either the number of bits in SDR N is too small to encode the required range or the number of required active bits W is too large compared to the number N.*

This can form ranges like 0-12, 10-22, 1000-1012 or any other range. The encoding value has no absolute meaning at the encoding stage and is used later as a spatial input non-supervised. Values encoded this way can represent any sensory input. For example, it can be the frequency the cochlea receives in the ear, a colour range, temperature, ultrasonic frequency or even a range of numbers received by some device representing a new kind of sensor.

All encoded values in this example overlap in two bits. If the greater width were used, the number of uniquely encoded values would decrease according to equation (19). However, suppose the number of required encoding scalar values defined by the range [min, max] is greater than the number of maximal possible uniquely encoded values Emax. In that case, the encoder will use the same encoding for multiple values from the range. Such configuration would prevent the HTM from distinguishing different values with the same code, which leads to improper learning. Such an SDR is shown in Figure 22 C. Therefore, the encoding range should be shared appropriately across available bits N with the given encoding width W.

One of the essential requirements in this research is the calculation of similarities between SDRs. Comparing several similarity functions found that typical sparsity (i.e., 2% - see 3.1.1) highly influences the similarity due to the significant difference between the number of non-zero bits and zero-bits. For this reason, the similarity function used in this work encounters only non-zero bits.

Given two arrays $A_1$ and $A_2$ with indexes of non-zero (active) cells $a_{11}, a_{12,..}, a_{1N1}$ and $a_{21}, a_{22,..}, a_{2N2}$ from arrays $A_1$ and $A_2$, respectively, the similarity $\boldsymbol{s}$ of arrays can be computed as shown in the following equation:

$$\boldsymbol{s} = \frac{\mathbf{100}}{\boldsymbol{max}(N_1, N_2)} \sum_{\boldsymbol{i=0}}^{\boldsymbol{min}(N_1, N_2)} a_1 a_2(\boldsymbol{i}) \qquad (20)$$

The equation does not propose strict limitations for the length of comparing array. However, all experiments in this work use arrays with the same length that differ in the position of active (non-zero) bits. Table 2 shows six examples that demonstrate how encoder-produced SDRs are compared.

| **Arrays** | **Similarity** |
|---|---|
| 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,<br>0, 0, 0, 0, 1, 1, 1, 1, 0, 0, | 100.0% |
| 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,<br>0, 0, 0, 0, 0, 1, 1, 1, 0, 0, | 75.0% |
| 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,<br>0, 0, 0, 0, 0, 1, 1, 1, 1, 0, | 75.0% |
| 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,<br>0, 0, 0, 0, 0, 0, 0, 1, 1, 0, | 25.0% |
| 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,<br>0, 0, 0, 0, 0, 0, 0, 1, 0, 0, | 25.0% |
| 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,<br>0, 0, 0, 0, 0, 0, 0, 0, 1, 0, | 00.0% |

*Table 2.*
*Similarities between several SDR examples*

The left column shows two arrays and the right column shows the similarity calculated by equation (20). Two arrays are similar if both have an active bit at the same index.

In this implementation, the HTM SDRs generated by encoders are used explicitly as an input of the Spatial Pooler, as described in the following section.

## 5.6 The definition of the model

The implementation of the HTM in this work models HTM entities in an object-oriented manner. This section describes the model of a few most important entities used in this research. It explains how the design decision fits the biology findings described in Chapter 2 and the overall HTM idea described in Chapter 3.1. Please see (Dobric D. , NeoCortexApi Entities, 2019) for a detailed implementation of all used entities.

The following entities (see Figure 23) and their relations have been defined to support the model defined in section 5.3: Cell, Synapse and Column (means mini-column as described in 2.3). The neural cell (see 2.1, 3.1.2) consists of a set of distal dendrite segments (see 2.5) and a set of *Receptor Synapses*. *Receptor Synapses* connect the cell as a source cell to distal dendrite segments of other cells, as shown in Figure 10 B. Please note that the *Cell* has no connection to the *ProximalDendrite*. This is because the cells in the HTM do not form direct connections to sensory cells. Instead, the connections to sensory in HTM cells are established from the *Column* entity (mini-column) through the *ProximalDendrite*. *For* this reason, the entity *Column* holds a *ProximalDendrite* as a property.

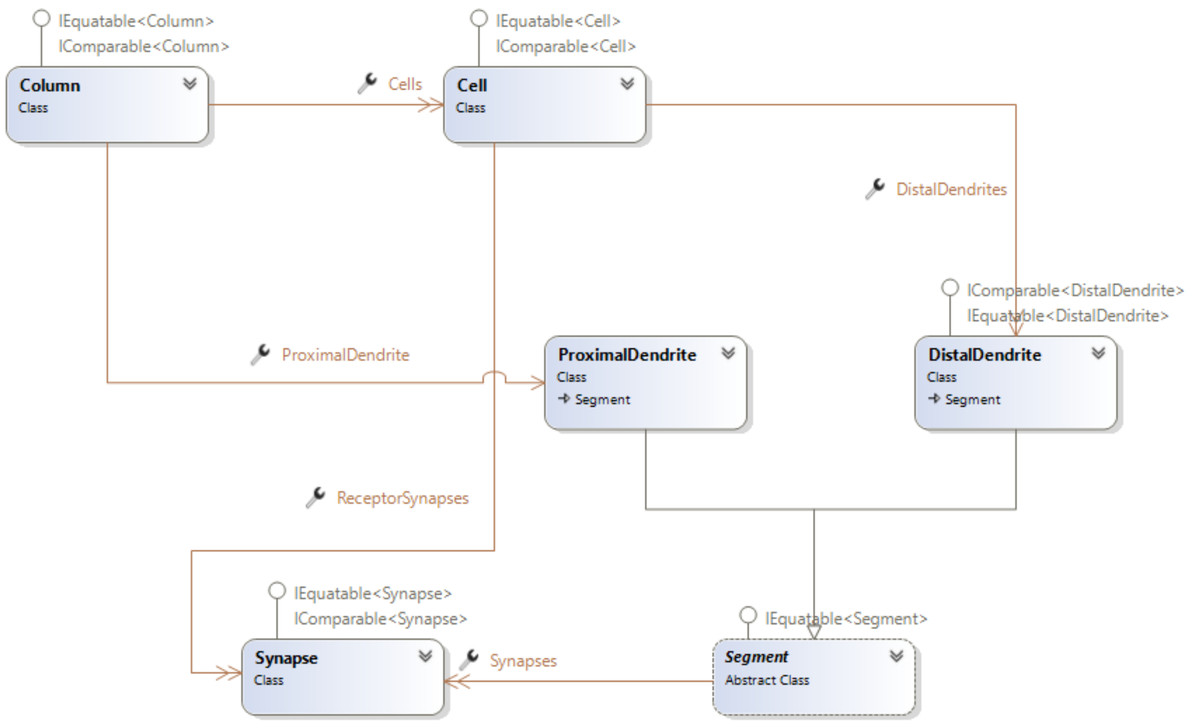


*Figure 23*

*The simplified HTM model contains the most important entities.*

The *ProximalDendrite* creates the pool of synapses and builds the *Receptive Field* to sensory cells, as described in sections 2.3 and 3.1. The class Segment is the simple base class implementation for all dendrite segments, as described in 2.5.

Another vital entity in the framework is the class *Connections*. It implements all the internally required states that form the Hierarchical Temporal Memory with Spatial Pooler and Temporal Memory.

## 5.7 Design and implementation of the Spatial Pooler

As already mentioned, the SP algorithm encodes the spatial pattern into the SDR code represented as the set of active columns. This section describes how this encoding in implementing the HTM in this research works.

To initialize the SP algorithm, the set of configuration parameters (*HtmConfig*) and the instance of the *Connections* entity (see the previous section) are required. During the initialization process of the SP, the *ProximalDentrite* segment creates potential synaptic connections to the set of sensory cells (see Figure 8). With these connections, the sensory input generated by an encoder is feed-forwarded into the HTM. The internal state of the algorithm is saved to the entity *Connections*. Once the synaptic connections from mini-columns to the input cells are established, they will change their permanence over time. Still, no new connections will be created, and no existing connections will be removed. With this decision, the SP does not generally support structural plasticity (see 3.1). As described later in the next section, this plasticity is a part of distal dendritic activation. The SP limits the permanence value to 1 and sets it to zero if it is under the defined value *SynPermTrimThreshold*. Currently, there is no biological evidence for this value, but all experiments in this work initiate this threshold to 0.05.

The input of the SP is an array of bits defined as a set of $N$ neurons $I^n$. This input can be multidimensional (see Figure 11), but internally it is flattened and then mapped to sensory neurons. Every input is represented as a set of values (0/1), where $N$ is the total number of neurons in the input. A flattened version of an input vector $I_k$ is defined as:

$$I_k = \{0,1\}^{1\text{x}N} \tag{21}$$

Also, a mini-column set $C_x$ in area x is defined as a flat column array, where $M$ is the total number of mini-columns:

$$C_x = \{C_1, C, .., C_M\}^{1\text{xM}} \tag{22}$$

Sometimes the index notation is omitted, and the mini-column set is declared as $C$. As mentioned in section 5.6, mini-columns connect to input neurons via the *ProximalDentrite* segment and form potential synapses. This happens at the initialization time of the SP. Connected synapses will not be destroyed, or a new synaptic connection will be created during the entire lifetime of the SP. The SP connects to a strictly controlled subset of input neurons defined by a receptive field (RF) of potential synapses of the mini-column. Giving the possible connection space $\lambda^{N\text{x}M}$ the following expression defines the set of potential synapses that build the RF to input neurons connected to the mini-column with index $k$.

$$X_k{}^{1\text{x}N} = \{ x_{k1} .., x_{kN} \} \mid X_k \in \mathrm{X}^{N\text{x}M}, 0 \leq k < M, x_{ki} \in \{0,1\} \qquad (23)$$

If the input neuron with index $i$ is synaptically connected to the mini-column $k$, the element $x_{ki}$ *from* $X_k$ is set to 1. Mathematically the SP algorithm maintains a matrix $\boldsymbol{\lambda}$ called in this work *connection matrix*. This matrix is the union of all receptive fields $X_k$.

$$\boldsymbol{\lambda} = \begin{pmatrix} x11 & x12 & \dots & x1N \\ x21 & & & \\ \dots & & & \\ xM1 & xM2 & & xMN \end{pmatrix} \Bigg| x_{ij} \in \{0,1\} \qquad (24)$$

Indexes $i$ and $j$ define the position of the flattened versions of columns and sensory neurons, respectively. The mini-column $C_i$ is potentially connected to the sensory neuron $I_j$ if $x_{ij} = 1$. Similarly, to $X_k$ the set of potential synapses of the mini-column $k$ can is defined as follows:

$$P_k = \{ p_{ku} .., p_{kw} \} \mid P_k \in \mathrm{P}^{N\text{x}M}, 0 \leq k < M, 0 \leq p_{ki} \leq 1, \in \{0,1\} \mid \qquad (25)$$
$$u, w \in \{0, N\}$$

## 5.7.1 The Overlap function

The core of the learning process in HTM is calculating the *overlap* between every mini-column and the currently presenting input pattern. The overlap algorithm aligns with equation (11) and calculates the overlap value for the entire mini-column set of the cortical area (5.3), as shown in Algorithm 1.

Algorithm 1 Overlap Calculation

```
01| **function overlap** ($I$, $\theta_c$)
02|     // $\theta_c$ Connected permanence threshold
03|     **FOREACH** col **IN** $C^{1xM}$ // traverse all mini-columns
04|             $o_k \leftarrow$ columnOverlap($I$, col, $\theta_c$)
05|     **ENDFOR**
06| **end**
07| // I: Input vector. k: mini-column to calculate the overlap.
08| **function columnOverlap**($I$, $k$, $\theta_c$)
09|         // gets the RF of column k.
10|         ${X_k}^{1xN} \leftarrow X$
11|         // Multiply every RF-bit with a connected synapse with the value of the input vector
12|         $o \leftarrow \{\sum x_{kj}\ i_j\} | x_{kj} \in X_k; p_{kj} \geq \theta_c, k \in \{1, M\}, i_j \in I, j \in \{1, N\}$
13|         **return** o
14| **end**
```

The overlap is calculated in every cycle for all mini-columns in set $\mathrm{C}$ (see lines 03-05). The overlap for every mini-column is calculated in the function *columnOverlap* (lines 08-13). The receptive field $X_k$ between the mini-column $k$ and the input vector, is retrieved from set $\mathrm{X}$ of all RFs in $\lambda$. Every $x_{kj}$ has an associated synaptic permanence value $p_{kj}$. This value controls if the potentially connected mini-column ($x_{kj} = 1$) will be included in the overlap. The function builds the sum of products $x_{kj} i_{j,}$ which encounters all mini-columns that are connected ($x_{kj} i_j | p_{kj} \geq \theta_c$) to the currently active input neurons.

The overlap algorithm is a straightforward and efficient one. Because of its efficiency, it can easily be applied even in cortical areas with a high number of neurons.

### 5.7.2 Learning patterns

During the learning process, the SP first calculates the overlap of the mini-columns set at the given cycle for the given input. Then, every time an input is presented to the SP, the set of potential synapses *Xk* defined by equation (23) is considered. Figure 24 shows the receptive field (simplified) of two columns in the cortical area of seven columns.

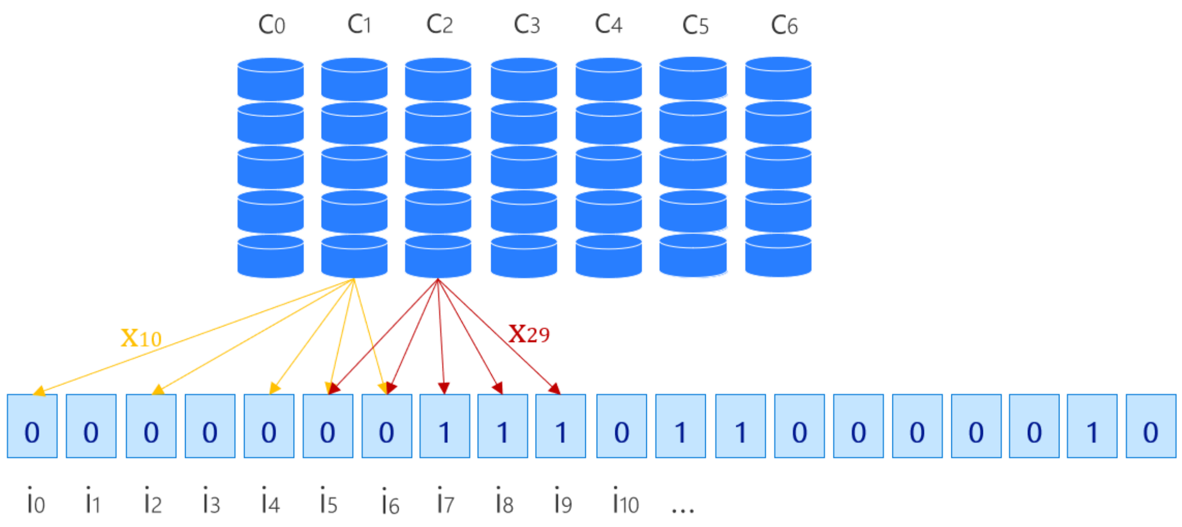


*Figure 24*

*Synaptic plasticity in the Spatial Pooler learning process.*

At some time *t*, the vector *It* is presented to the SP. As shown in Algorithm 2 (line 07), the SP first calculates the overlap of every mini-column by using Algorithm 1. In this example, the column's $C_1$ receptive field $X_1$generates the overlap 0, because none of input neurons in the field *i0,.., i6*is active. The overlap of the column $C_2$ that spans input neurons *i5,.., i9* is 3, because 3 of input neurons in the receptive field are active.
Next, the permanence of all synapses in the receptive field $x_k$ every mini-column is adapted (line 9). The adoption process biologically represents synaptic plasticity, as described in sections 2.1 and 5.2. In this step, the permanence values $p_{ij}$ defined by (25) of all synapses within the receptive field $X_k$ are incremented with the value *actInc* if the synapse connect to the active (firing) inut neuron. In contrast, the synaptic permanences are decremented by the value *inactDec* if connected to non-active input neurons. Decrement of the permanence value of the synapse is a natural biological way to integrate the error feedback, similar to backpropagation.
In the same turn, the algorithm forms the set of active mini-columns $C^a$ that have an overlap higher than the *stimulus threshold* $\theta_p$ for the given input (line 11). Please note that this is the simplified version of learning without inhibition and the homeostatic plasticity mechanism described later in this chapter. Learning patterns is an iterative process that typically takes many cycles. The pattern is, by definition, learned if the output SDR calculated by the Spatial Pooler function *sp* does not change over time when the same pattern is presented to the SP.

Algorithm 2. The basic algorithm for learning spatial patterns

```
01| function compute ($I, \theta_p, \theta_c, actInc, inactInc$)
02|   // $\theta_p$: Stimulus Threshold. Num. of connected synapses to activate the column
03|   // actInc   : increment value for active column
04|   // incatInc : increment value for active column
05|   // I : Input vector
06|   FOREACH col IN $C^{1xM}$ //traverse all mini-columns
07|       $o_k \leftarrow$ columnOverlap(input, col, $\theta_c$)
08|       // increment the permanence if the column is connected to active neuron.
09|       $p_{ki} \leftarrow p_{ki} + \delta \mid \left\{ \begin{matrix} \delta = actInc; & x_{kj} * i_j = 1 \\ \delta = -1 * inactDec; & x_{kj} * i_j = 0 \end{matrix} ; x_{kj} \in X_k, i_j \in I, k \in \{1, M\} \right\}$
10|       // Activate all columns with overlap higher than stimulus threshold.
11|       $C^a \leftarrow \{c_k \mid o_k \geq \theta_p, k \in \{1, M\}\}$
12|   ENDFOR
13| end
```

Mathematically, according to (9), the set of active mini-columns $C^a$ converges to the stable state. The stability is defined by the following equation.

$$C^a = \mathrm{sp}(I) = const;\ t_i > t_s \tag{26}$$

The SP is stable if the set of active columns $C^a$ , learned for every input presented to the SP, does not change in any iteration after the iteration $ts$. In this case, the iteration $ts$ is the iteration (cycle) of entering the stability.
In real-world applications, it will not be possible to ensure that SP will learn the entire possible input set in all scenarios. However, learning many patterns will ensure uniform activation of all mini-columns, leading to better stability.

Stability issues of the original version of the SP have been discovered and, in detail, analysed in this work. In addition, the current SP algorithm has been extended and improved. More information about stability and algorithm improvement can be found in Chapter 8.

### 5.7.3 Column Inhibition

Another vital part of the SP is the inhibition algorithm, which represents the biological inhibition process described in sections 2.1 and 2.4.
The inhibition is the process of deactivating the fraction of mini-columns that tend to be active in the current learning cycle. This ensures that the encoding of learned

patterns becomes sparse and not dense. In this work, several versions of the inhibition have been implemented. However, two diametral approaches are considered in this section: global and local inhibition. The global inhibition performs the active mini-column deactivation in the entire mini-column space defined by equation (22). In contrast, the local inhibition inhibits columns in the locally defined neighbourhood.
The inhibition algorithm implements the inhibition mechanism described in 2.1. The HTM model described in section 5.3 relies on the explicit use of excitatory cells in mini-columns. However, following the biological mechanisms described in Chapter 2, there must also be a set of inhibitory cells that prevent the overexcitation of mini-columns. The design of the HTM in this work omits using explicit inhibitory cells for performance reasons. As a replacement for inhibitory cells, the HTM in this work provides an inhibition algorithm that extends Algorithm 2 by inhibiting mini-columns activated in line 11.

#### 5.7.3.1 Global Inhibition

The main idea of inhibition is to inhibit potentially active mini-columns in an area occupied by the so-called *Inhibition Radius*. In the case of *Global Inhibition*, the *Inhibition Radius* is calculated as the maximum number of cells of a single dimension across column dimensions (see also equation (12).

$$ir = \max\left(\{cdim_1, cdim_2, .., cdim_{cd}\}\right) \tag{27}$$

Value $cdim_i$ is the number of mini-columns in the $i$-th dimension of available $cd$ dimensions. If the mini-column space is single-dimensional, this value equals the number of mini-columns $M$ in the HTM area as defined by equation (22). In the case of multi-dimensional mini-column space and global inhibition, the *Inhibition Radius* $ir$ is the maximum number of mini-columns across all dimensions.

Algorithm 3 shows how mini-columns are globally activated. First, the inhibition radius is calculated by equation (27) in line 9. As next, the algorithm needs to determine the inhibition density. The density is defined as the percentage of mini-columns allowed to be activated inside the *inhibition radius*. According to the previous discussion (see 3.1.1), the number of active mini-columns per inhibition area (*NumActiveColumnsPerInhArea)* is usually around 2% of the total mini-columns. This value can be set by the configuration parameter $\mathrm{actCols}$. The number of active mini-columns can also be controlled by the configuration property *LocalAreaDensity* (see HtmConfig in section 5.3). It can be either manually set or calculated, as shown in line 13.

If *LocalAreaDensity* is not specified, the calculated density is limited to 50%, assuming higher density is not sparse. Also, if *LocalAreaDensity (*locDense*)* is not set, then the number of active columns in the inhibition area $actCols$ must be set.

$$d = \min\left(0.5, \frac{actCols}{\min\left(tnc,\ (2ir+1)^{cd}\right)}\right) \qquad (28)$$

The density is a fraction of the wanted number of active columns (actCols) and the minimum of the total number of mini-columns (tnc), and the
inhibition diameter $2ir + 1$ powered with the number of mini-column dimensions (cd). This expression is calculated in line 11. The overlap function in line 15 calculates ac mini-columns that can be activated. To prevent overactivation, the inhibition algorithm uses the density (line 17) to randomly select the top active mini-columns utilising the function $\mathrm{T}(ac)$.
In the final implementation of the SP, the calculation of the value *ac* (active mini-columns calculated by overlap) is not a part of the inhibition algorithm.
It is instead used as the input argument of the inhibition function. It is added to Algorithm 3 to get a better understanding of the dependency between overlap, active mini-columns and inhibition.

Algorithm 3. Global Inhibition over the entire HTM area.

```
01| function inhibitGlobal (cdims←$\{cdim_1, cdim_2, .., cdim_{cd}\}, locDens, actCols, \theta_p, i$)
02|  // cdims: Number of columns in every column dimension
03|  // locDense: Configured Local Aread Density
04|  // actCols: Number of active columns. Defines the sparsity.
05|  // $\theta_p$: Stimulus Threshold
06|  // i: The input pattern that is currently learned.
07|
08|  // calculate the inhibition radius
09|  $ir \leftarrow \max(\mathrm{cdims})$
10| // Calculates the total number of mini-columns.
11| $tnc \leftarrow \prod(dims)$
12|  // calculate local area density if not explicitly set
```

```
13| $d \leftarrow \left\{ \begin{matrix} locDense;\ locDense > 0 \\ locDense = \min(0.5, \frac{actCols}{\min(tnc, (2ir+1)^{cd})}) \end{matrix} \right\}$
14| // Activate all columns with overlap higher than stimulus threshold.
15| $ac \leftarrow \{c_k \mid columnOverlap(I,k) \geq \theta_p, k \in \{1,..,\mathrm{tnc}\}\ \}$
16| // The inhibition. Select top mini-columns and make the result sparse.
17| $C^a \leftarrow \mathrm{T}(ac); |C^a| = d\ tnc$
18| **end**
```

For example, assuming that the number of mini-columns is 10 (see Figure 25), the stimulus threshold $\theta_p$=*2* and *actCols=4*, the inhibition algorithm would first select six mini-columns (black and green) in line 15 and then adjust this value in line 17 using the calculated density (green only). The final number of active mini-columns would be four, as shown in the figure.

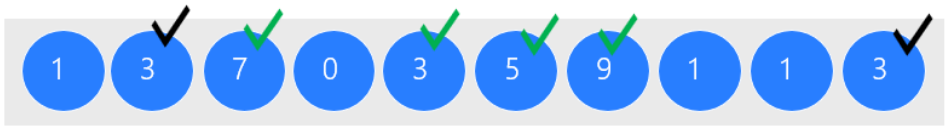


*Figure 25*

*Global Inhibition example. Columns with the maximum overlap are selected as active.*

#### 5.7.3.2 Local Inhibition

Nowadays, it is unclear how inhibition in the neocortex is exactly performed. For this reason, in this work, several algorithms were tried. One of them is the local inhibition algorithm shown in Algorithm 4. Algorithm 4 requires as input the exact topology of the HTM area defined by column dimensions *cdims* and input dimensions *idims*.
Both parameters specify how many mini-columns and input neurons are placed in every dimension to form the HTM area. However, in contrast to the global inhibition

Algorithm 3, local inhibition, has a more complex calculation of the inhibition radius (lines 08-14). First, the input span $s$ is calculated as a ratio of the number of input neurons in the receptive field of the mini-column and the number of mini-columns (line 09). Then the average of the column-input ratio across all dimensions is calculated (line 12). Next, these two values are used to calculate the inhibition radius of the columns' inhibition neighbourhood (line 14).

In contrast to global inhibition, the local inhibition algorithm traverses all mini-columns in the HTM area (lines 19-25) and activates mini-columns only inside the column’s neighbourhood. The function *getColumnNeighborhood* selects all mini-columns inside the neighbourhood of the currently calculating mini-column $k$. Columns selected by this function have overlap greater than the stimulus threshold $\theta_p$ and the overlap of the currently calculating column.

Figure 26 shows two examples of local inhibition. The grey frame represents the sliding window (local inhibition area) with the calculating (centre of the local inhibition area) mini-column marked by the arrow. The inhibition radius defines the local inhibition area or sliding window. For $\theta_p = 2$, all mini-columns inside the window are marked if their overlap is greater than two and the overlap of the calculating mini-column marked with the arrow.

Algorithm 4 Local Inhibition algorithm

```
function inhibitLocal ($C_x$, cdims←$\{cdim_1, cdim_2, .., cdim_{cd}\}$,
                       idims←$\{cdim_1, cdim_2, .., cdim_{cd}\}$, $locDens, actCols, \theta_p, i$)
 // $C_x$: Mini-Columns space in calculating area x.
 // dims: Number of columns in every column dimension
 // locDense: Configured Local Area Density
 // actCols: Number of active columns. Defines the sparsity.
 // $\theta_p$: Stimulus Threshold
 // i: The input pattern that is currently learned.
 // calculate the average input span across all receptive fields in $X$.
   $s \leftarrow \left\{ \frac{\sum len(X_k)}{M} \right\} \mid X_k \in X$

 // calculate the average column-input ratio across dimensions
   $f \leftarrow \left\{ \frac{\sum \frac{cdim_i}{idim_j}}{M} \mid i \in \{1, \mathrm{cd}\}, j \in \{1, \mathrm{id}\} \right\}$

 // calculate inhibition radius
 $ir \leftarrow \max(1, \frac{(f*s)-1}{2})$
 // Calculates the total number of mini-columns.
 $tnc \leftarrow \prod(dims)$
 // calculate local area density if not explicitly set
 $d \leftarrow \left\{ \begin{matrix} locDense;\ locDense > 0 \\ locDense = \min\,(0.5, \frac{actCols}{\min\,(tnc, (2ir+1)^{cd})} \end{matrix} \right\}$

 // Traverse all columns and calculate inside the sliding window
  FOREACH $c_k$ in $C_x$
    // Get columns in the neighbourhood (sliding window)
    // that has overlap higher than the current mini-column
     highCols ← $getColumnNeighborhood(k, ir, \theta_p, i)$
    // Append mini-columns until $d * tnc$ columns are activated
     $C^a \leftarrow append(c_k\ ; c_k \in$ highCols ; $|C^a| \leq d\ tnc)$
  ENDFOR
end

// Gets mini-columns from the neighbourhood with overlap $\geq \theta_p$
//  and overlap greater than the calculating mini-column k.
function getColumnNeighborhood ($k, ir, \theta_p, i$)
 // Get all columns in the area with overlap higher than stimulus threshold.
 $C^a \leftarrow \{c_k \mid columnOverlap(i,j) \geq \theta_p\ \wedge$
           $columnOverlap(i,j) > columnOverlap(i,k));$
           $j - ir \leq j \leq j + ir\ \}$
 $return\ C^a$
end
```

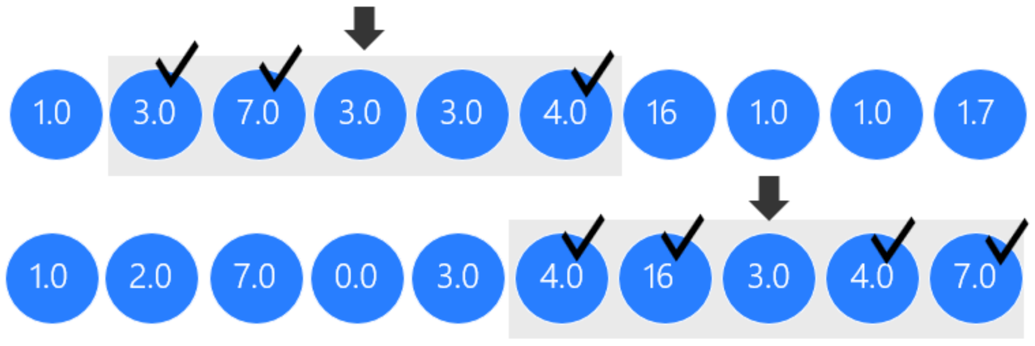


*Figure 26*
*The local Inhibition moving window defines the local inhibition area.*

#### 5.7.3.3 Summary

In this research, multiple different versions of the inhibition algorithm were tested. However, significant differences in learning performance could not be detected. The global inhibition supported a faster calculation cycle and was used in most experiments in this work. In contrast, the local inhibition is CPU intensive because it has to create local areas and shift them over the entire mini-column space. This algorithm aims to ensure balanced network excitation and inhibition, as mentioned in Chapter 2. Neurons used in the cortical algorithm in this work are responsible for the excitation.
In contrast, the Inhibition algorithm ensures that only a sparse set of neurons remains activated in each cycle. Without inhibition, the learning algorithm would, over time, activate most mini-columns, preventing it from learning. This is why an inhibition algorithm is critical in CLA. However, nowadays, it is not clear how inhibition exactly works in the neocortex. Nevertheless, local inhibition seems to match well with the biological findings. The current implementation is just one of many possible implementations that keep the cortical algorithm stable and ensure sparse encoding.

Improving and investigating inhibition is the part of future work.

### 5.7.4 Computational Design of Homeostatic Plasticity

Similar to mini-columns inhibition, an active biological homeostatic plasticity mechanism is described in 2.6 that takes control of a uniform excitation of neurons. However, in contrast to inhibition, this mechanism regulates the excitability of neurons relative to their activity in the defined cortical area. The Cortical Algorithm in this work provides an algorithm inside the SP as a computational equivalent of homeostatic plasticity.
This algorithm consists of two parts: *Synaptic boost of mini-colum*ns *with low overlap* and *uniform activation* of mini-columns.

#### 5.7.4.1 Synaptic Boost of mini-columns with insufficient proximal connections

A mini-column has a low overlap if the total count of synapses that have established connections at the proximal dendrite segment is below the *Stimulus threshold* $\theta_p$ (5.4), in a learning cycle. If the mini-column has a low overlap, then the permanence values of all potential synapses of a mini-column will be slightly incremented by parameter *si* (*Stimulus Increment*). As described in 5.7.2, independent of boosting, synaptic permanences are mainly incremented if connected to firing neurons.

Algorithm 5 is an extended version of basic spatial learning shown in Algorithm 2. It refactors and extends the basic idea of learning by adding inhibition and synaptic plasticity in lines 09-15. Also, the original synaptic learning from Algorithm 2 in Line 09 is moved to the function *adaptColumnSynapses* Line 17 and extended with the synaptic boosting mechanism of inactive mini-column in the function *boostSynapses* Line 27.

After the permanence values are adapted (Line 21), the permanences of mini-column synapses connected to firing input neurons with low overlap (Line 30) will be slightly increased (Line 32). It is crucial to choose the value $\theta_p$ carefully, concerning the number of input bits N and the potential radius described in equation (16). In most experiments in this work, the value for $\theta_p$ was typically chosen between 20-50% of the number of input bits N. Choosing higher or lower values prevents the SP from converging to the stable state defined in 5.7.2. In this work, no further investigation of this threshold has been done. However, it is still an important part of the research because it significantly influences learning patterns in the SP.

Algorithm 5 Spatial learning algorithm extended with a synaptic plasticity.

```
function compute($I$, $si$, $\theta_p$,$\theta_c$, $si$)
  // $I$ :        Input Vector
  // $\theta_p$:       stimulus threshold. Required num of connected synapses.
  //actInc:    increment value for  active column
  //inactInc:  increment value for  active column
  // $\theta_c$  connected permanence threshold
  // si:  stimulus increment
  // traverse all mini-columns
  FOREACH $col$ IN $C^{1xM}$
      $o_k \leftarrow$ columnOverlap(input, col)
      $C^a\ (k) \leftarrow inhibit(..)$
      $C^a\ (k) \leftarrow adaptColumnSynapses(k, o_k, \theta_p)$
    ENDFOR
  return $C^a$
end

function adaptColumnSynapses($k$, $o_k$, $\theta_p$)
  // st: Stimulus Threshold.  $p_{ki}$: Permanence of synapse of $c_k$  with input i.
  // k: The index of the calculating active mini-column.
  // increment the permanence if overlap greater than stimulus threshold.
  $p_{ki} \leftarrow p_{ki} + \delta \mid \left\{ \begin{matrix} \delta = actInc; & \sum(o_k \geq \theta_p) \\ \delta = -1 * inactDec; & \sum(o_k < \theta_p) \end{matrix} ; i \in \{1, N\} | I(i) = 1, k \in \{1, M \right\}$

  $boostSynapses(I, k, si, \theta_c, \theta_p)$

end
// For the full implementation. See BoostProximalSegment (Dobric, 2019)
function boostSynapses(I, $k$, $si$, $\theta_c$,$\theta_p$)

  // Boost synapses of active mini-columns with no enough connected synapses.
  WHILE($\sum(p_{ki}) \leq \theta_p | \ p_{ki} \geq \theta_c, p_{ki} \in P_k$)
    // increment the permanence if overlap greater than stimulus threshold.
    $p_{ki} \leftarrow p_{ki} + si \mid i \in \{1, N\} | I(i) = 1, k \in \{1, M\}$
   END
 end
```

#### 5.7.4.2 Uniform Activation of mini-columns

The second part of the plasticity (see 2.6) implementation ensures that all mini-columns in the HTM area become uniformly activated. As described later in Chapter 8, the absence of this kind of plasticity leads to uncontrolled sparsity in the cortical area, leading to incorrect prediction and inaccurate learning. *The columnar overlap* and *columnar activation frequency* are considered to ensure the uniform participation of mini-columns in learning. Assuming that the value of synaptic permanence defines some kind of energy accumulated in the synaptic connection, the goal here is to keep that energy uniformly distributed across the entire cortical area. For example, inactive mini-columns must be stimulated (boosted) compared to their neighbours. That means their overlap values to input neurons will be slightly increased by a column boost factor $b_k$, which needs to be recalculated in every cycle. If the mini-column is not active enough, it will often not participate in learning. Because of that, to activate the mini-column, the column overlap needs to be controlled too. Having the set of events $\mathcal{F}$ the event $f_k \in \mathcal{F}, f_k \in \{0,1\}$ is counted every cycle. The set $\mathcal{F}$ holds elements appearances (0 or 1) of the event $f_k$ for every mini-column *k* in the current cycle. With this in mind, two such event types are considered: Participation of the mini-column in the overlap and activation of the mini-column. Both events are calculated with the help of the same equation shown in Algorithm 6.

Algorithm 6 Calculation of the normalized frequency of an event

```
01| function calcEventFrequency($C^f, \mathcal{F}, p, i$)
02|   // $C^f$ : The current frequency values of the event.
03|   // $p$ : Activation period value.
04|   // $i$ : The learning cycle (iteration)
05|   // $\mathcal{F}$ : List of any kind of event {0,1}

06|      $p = \begin{cases} p; i < p; \\ i; i \geq p \end{cases}$
07|   // Calculates the normalized event frequency.
08|      $c_k^f \leftarrow \frac{(p-1)\, c_k^f + f_k}{p} \mid k \in \{1, M\}, f_k \in \mathcal{F}, f_k \in \{0,1\}, c_k^f \in C^f, 0 \leq c_k^f < 1$
09| end
```

The parameter *p* is the period value that defines how many cycles the calculated frequency will be held on low values (line 06). The equation in line 08 calculates in the normalized form the frequency of the appearance of events in the event set $\mathcal{F}$ during the learning process of the spatial pattern. The set $C^f$ defines the current state of normalized frequencies $c_k^f$ of the event $f_k \in \mathcal{F}$ for every mini-column $c_k \in \mathrm{C}$.

**The Columnar Overlap Frequency**

With a given frequency calculation of an event, the mini-column with low overlap frequency will be reactivated (boosted) by using Algorithm 7. First, the overlap events are collected in every cycle (Line 08). The column is considered in this calculation if it has any overlap $o_k > 0$ as described in 5.7.1 and calculated by Algorithm 1. Elements $f_k \in \mathcal{F}$ hold in this case overlap frequency events, with non-zero if the mini-column has an overlap and with zero if the mini-column has no overlap in the current cycle. The new state of overlap frequencies $C^{fo}$ is calculated in line 10 from the current frequency state $C^{fo}$ and the set of events $\mathcal{F}$.

Algorithm 7 Boosting of mini-columns with low overlap frequency

```
01| **function BoostByOverlapFrequency**($o, C^{fo}, p, f^{omin}, i, s$)
02|   // $o$: Overlap calculated for all mini-columns ; $o \in O^{1xM}$.
03|   // $C^{fo}$ : The current state of mini-column overlap frequencies.
04|   // $p$ : The period value. $s$: Increment value
05|   // $f^{omin}$ : The minimum required overlap frequency. $0 \leq f^{omin} < 1$
06|   // $i$: The learning cycle
07|   // Given a frequency set $\mathcal{F}$ ; $f_k = 1$ if the mini-column has an overlap.
08|   $f_k \leftarrow \left\{ \begin{cases} 1; o_k > 0; \\ 0; o_k = 0; \end{cases} \mid k \in \{1, M\}, f_k \in \mathcal{F} \right\}$
09|   // Calcuates the overlap frequency.
10|   $C^{fo} \leftarrow calcEventFrequency(C^{fo}, \mathcal{F}, p, i)$
11|   // Get the subset of columns to be considered.
12|   $C'^{fo} \leftarrow \left\{ \begin{cases} C^{fo} & ; globalinh. \\ \{c_i^{fo}\} \mid |(k-i)| < ir; local\ inh.; \end{cases} \mid k \in \{1, M\}, \{c_i^{fo} \in C^{fo} \right\}$
13|   // Calculates the minimum required overlap frequency in the inhibition area.
14|   $c^{fomin} \leftarrow (\max(C'^{fo})\ f^{omin}) \mid k \in \{1, M\}, c_k^{fomin} \in C^{fomin}$
15|   // Create the set of mini-columns with the low overlap frequency
16|   $c_k^{low} \leftarrow \{c_k\}$ ; $c_k < c^{fomin}$ ; $k \in \{1, M\}, c_k^{low} \in C^{low}$
17|   // Boost synapses of all mini-columns with the low overlap-frequency
18|   **FOREACH**($p_{ki} \mid p_{ki} \in P_k^{low}$)
19|     // increment the permanences of mini-columns with low overlap.
20|     $p_{ki} \leftarrow p_{ki} + s \mid i \in \{1, N\} | I(i) = 1, k \in \{1, M\}$
21|   **END**
22| **end**
```

The required minimum overlap frequency is controlled by the configuration parameter $0 \leq f^{omin} \leq 1$. The minimum required overlap frequency for every mini-column is calculated in line 14 as a multiply between factor $f^{omin}$ and the current maximal overlap frequency inside the neighbourhood defined in line 12. In the case of local inhibition, only neighbouring mini-columns are considered. This neighbourhood area is occupied by the mini-columns inside the inhibition radius $ir$. In the case of global inhibition, the entire column set is considered. Inside the selected area, mini-columns that have a low frequency are chosen $c_k^{low} \in C^{low}$ (line 16).

Finally, in the loop in lines 18-21, permanence values of mini-columns with low overlap frequency are incremented. The permanences of mini-columns with the low overlap frequency belong to the temporary calculated space $P_k^{low}$associated with the mini-column $k$.

**The Columnar Activation Frequency**

After the reactivation (boosting) of mini-columns by the low overlap, the mini-columns with the low activation frequency must also be reactivated, as shown in Algorithm 8. First, the mini-column activation frequency is calculated (line 08). The event appears (1) if the mini-column is active in the current cycle. This is the case when the number $\sum p_{ki}$ of connected synapses ( $p_{ki} \geq \theta_c$) is greater than the stimulus threshold $\theta_p$.The normalized frequency of the column activation is calculated in line 10 using Algorithm 6.

Algorithm 8 Boosting of mini-columns with low activation frequency.

```
01| function BoostByActivationFrequency($C^{fa}, p, f^{amin}, b_{max}, \theta_p$)
02|   // $C^{fa}$: The current state of mini-column activation frequencies.
03|   // $p$: The period value.
04|   // $f^{amin}$: The minimum required activation frequency. $0 \leq f^{omin} < 1$
05|   //$b_{max}$: Maximal boost value.
06|   // $\theta_p$: Stimulus Threshold. Num. of connected synapses to activate the column.
07|   // Given a frequency set $\mathcal{F}$; $f_k = 1$ if the mini-column is active.
08|   $f_k \leftarrow \left\{ \begin{matrix} 1 = \sum p_{ki} \geq \theta_p | p_{ki} \geq \theta_c \\ 0 = \sum(p_{ki}) < \theta_p | p_{ki} \geq \theta_c \end{matrix} | k \in \{1, M\}, f_k \in \mathcal{F}, p_{ki} \in P_k \right\}$
09|   // Calcuates the activation frequency.
10|   $C^{fa} \leftarrow calcEventFrequency(C^{fa}, \mathcal{F}, p)$
11|   // Get the subset of columns to be considered.
12|   $C'^{fa} \leftarrow \left\{ \begin{matrix} C^{fa} & ; globalinh. \\ \{c_i^{fa}\} \mid |(k-i)| < ir; local\ inh.; \end{matrix} | k \in \{1, M\}, \{c_i^{fo} \in C^{fo} \right\}$

13|   // Calculates the minimum required overlap frequency for the cycle.
14|   $c^{famin} \leftarrow \max(C'^{fa}) * f^{amin} | k \in \{1, M\}$
15    IF $|c^{famin}| > 0$
16|       $c_k^b \leftarrow \frac{(1-b_{max})\, c_k^{fa}}{c^{famin}} + b_{max} | k \in \{1, M\}, c_k^{fa} \in C^{fa}$
17|   ENDIF
18| end
```

The required minimum overlap frequency is controlled by the configuration parameter factor $0 \leq f^{amin} \leq 1$. Depending on the inhibition algorithm, the subset of mini-columns $C'^{fa} \in C^{fa}$ is considered (line 12). The minimum required activation frequency for every mini-column of the current cycle is calculated in line 12.

The boost value for the mini-column is calculated if the configuration value $f^{amin}$ is not set to zero (line 14). Finally, the equation in line 16 calculates the boost factors $c_k^b$ for every mini-column. The boost factors are used in the learning *compute* function in Algorithm 5 in line 10 by calculating the overlap. Every calculated overlap value $o_k$ is multiplied with the corresponding boost factor calculated in the previous cycle. With this change, Algorithm 5 completely supports the boosting described in this section.

### 5.7.5 Comparison to existing Sparse Encoding Algorithms

Most models described in the section (2.6) are motivated by the biological vision system of the brain. In contrast, the algorithm behind HTM - Spatial Pooler provides the general sparse encoding model that can be used for any input pattern. Depending on the point of view, it is crucial to understand that sparse encoding in this context can be classified into two categories: soft and hard-sparse. For example, models described in section 3.1.1, SparseNet and ICA produce *soft*-sparse code. It means they both produce neural activity with a narrower or (peakier) Gaussian distribution. In contrast, other models, like the Sparse-Set Coding network (SSC), produce a more efficient hard-sparse code (see 3.1.1). The Spatial Pooler belongs to the same category of hard-sparse coding algorithms.

As mentioned, the general motivation behind SP and HTM is to emulate the brain's reliance on binary data streams.Interestingly, the augmented version (3.1.1) of the Spatial Pooler modifies the notation of the overlap function. The original overlap function previously defined in (5.7.1) takes binary values 0 and 1, indicating whether the mini-column is connected to an active or inactive input. The modified Augmented SP violates the claim of the HTM theory that only binary activation is biologically plausible. This change might produce favourable results for a specific use case but should be approached differently.

As described later in Chapter 9, the colour of the pixel represents contextual information. Similarly, some other information like time delay between music notes can also be considered contextual information. Section 9.4 provides the exact definition of contextual information and proposes how it should be encoded.

Hofmann's Sparse Associative Memory (SAM) model, also mentioned in section (3.1.1) randomly connects input neurons with the neurons in the hidden layer. Hoffmann argues that connections in SAM would be biologically more plausible than the SP due to the fixed number of connections required by the SP algorithm. This claim is particularly incorrect because it assumes fixed connections between mini-columns and input neurons. The SP creates a random set of potential synapses inside the defined receptive field, which is created during the initialization process of the SP.

With this, it is not required to continuously recalculate the number of connections, which would be biologically implausible. Additionally, as briefly described later in this work, synaptic connections are controlled by several plasticity mechanisms aligned to biological findings.

### 5.7.6 SP Summary

This section summarized the most important details related to designing and implementing the Spatial Pooler algorithm used in this work. It described how the native overlap (5.7.1) function is used to implement the learning of spatial patterns by using a simple Hebbian learning rule (5.7.2). It also introduced many important parameters that control the learning process. By design of the cortical algorithm aligned with HTM, learning the pattern happens on the level of mini-columns and not at the cellular level inside mini-columns. The pattern is learned and recognized by the activation of mini-columns. The currently activated set of mini-columns encodes the input to the sparse representation. The sparsity is directly controlled by the inhibition (5.7.3) and boosting (5.7.4) algorithms that ensure that all parts of the area are uniformly used according to the biological homeostatic mechanism.

Figure 27 illustrates the learning process of the Spatial Pooler. The SP learns two spatial patterns represented as SDR. The input SDR in the example was created by the scalar encoder (5.7.5 for values 0 and 7. Encoding was configured to encode 100 scalar values as SDR with 200 bits and 15 non-zero bits. The following arrays represent positions of non-zero bits in the SDR for randomly chosen values 0 and 7, respectively.

**0:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
**7:** 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27

SDRs of named values overlap in two bits only at positions 13 and 14, indicating a slight similarity relative to the set of 100 values used by the encoder. The diagram in Figure 27 at the top shows active input neurons at the vertical axis (0-100) representing the encoded spatial input value, which the SP will learn. The learning experiment presents both values, one after the other, to the SP. Every time the SP sees some values, it will learn the pattern and activate a set of mini-columns representing the pattern. This process is repeated iteratively until the SP enters the stable state. By following Figure 27 – bottom, it can be noticed that the set of active mini-columns for both patterns change in the first five iterations. After the 6th iteration step (bottom left), the SDR enter the stable state for the value 0. The SDR of the value 7 enters the stable state at the 4th iteration step. As previously discussed, the SP continuously activates, reactivates and inhibits mini-columns in the entire area in every iteration step. In this context, estimating the exact step when a spatial pattern is learned is not of interest.

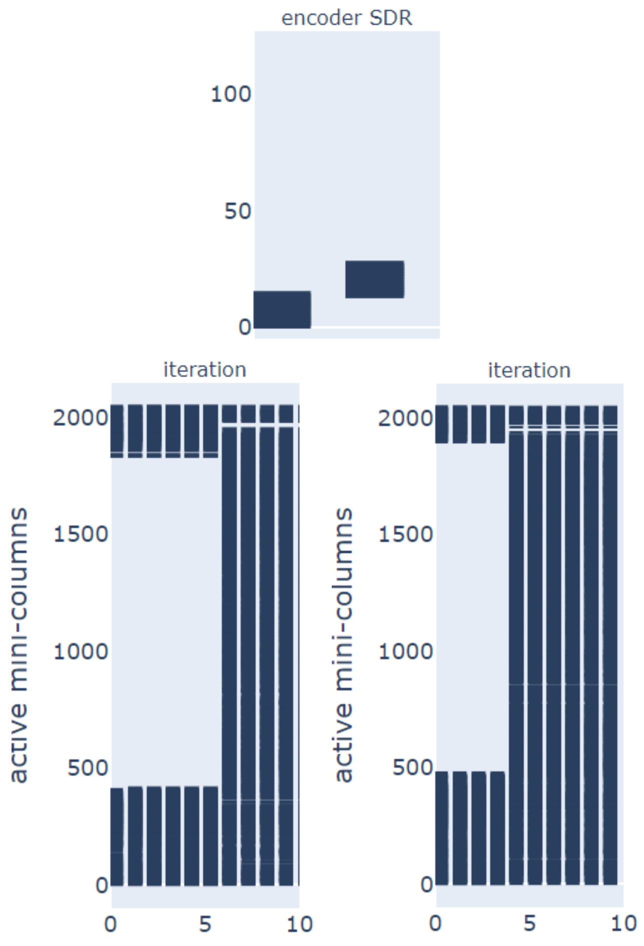


*Figure 27*

*Encoding of the spatial input to SDR. Top – The SDR of scalar values 0 and 7 encoded by the scalar encoder. Bottom left – active mini-column SDR of the value 0. Bottom right – active mini-column SDR of the value 7.*

All patterns enter the stable state at different iteration steps. Learning a pattern takes a short time, which means less than five cycles. However, when learning many patterns, the homeostatic plasticity algorithm described in 5.7.4 will actively reactivate mini-columns in every iteration to achieve the uniform distribution of permanence values (energy) across the entire synaptic space in the area. This process might take hundreds or even more iterations until all mini-columns become uniformly activated.

One of the essential outputs of this research, briefly discussed in Chapter 8, improves the homeostatic plasticity in the Spatial Pooler to achieve a stable learning process.

## 5.8 Neural Association Algorithm

This section describes how mini-column SDRs and the population of neurons, in general, can further be used to build associations between SDRs, establish semantic learning and create meaning. Such associations make semantic connections (dependencies) between SDRs representing contextual states. For example, if the SDR1 encodes visual information (context1) and SDR2 encodes audio information (context2), then the synaptic connection between two SDRs creates an association between contextual states encoded by SDRs. If the direction of the synaptic connection (association) does not matter, the associating context is called *Spatial Context*. In that case, State1 associates with State2 and vice versa. For example, hearing a song is often associated with the location and vice versa. In this work, the Spatial Context is created by the Spatial Pooler. However, some contexts may be represented by SDRs that are encoded by some other algorithms. For example, grid cells (Marianne Fyhn,Torkel Hafting,Menno P. Witter,Edvard I. Moser,May-Britt Moser, 2008) encode the location information, which defines a *Location Context* (Bennett, 2020).

If there is a causality between SDRs, then the association direction does matter. Because of the time dependency, such associations create the *Temporal Context*. Therefore, if State2 follows State1, State1 creates a *Temporal Context* for State2.

Section 5.8.1 describes a general model between mini-columns, cells, segments and synapses. Section 5.8.2 describes how learning associations between cell populations and a single segment might work. Section 5.8.3 describes in detail how the structural plasticity is modelled.

Finally, the learning process is split into sections 5.8.4 and 5.8.5. Section 5.8.4 describes how the cortical algorithm activates the population of cells (cellular activation algorithm). Section 5.8.5 describes which role play segments in the learning process. The *Neural Association Algorithm* (NAA) described in this chapter is a theoretical model that demonstrates how Cortical Learning might work. However, the central part of this algorithm, restricted to a single area, is implemented in this work as part of the previously introduced framework *NeoCortexApi* (Dobric, 2019) following the idea of HTM (3.1). This implementation was used in all experiments later described in chapters 6 to 8.

### 5.8.1 The proposed Model

After the SDR code has been generated, the association can be built between SDRs independent of their root. This critical claim makes it possible to combine sensory inputs from different sources and process all sensory inputs exactly the same way. Following this idea, like in mammals, the same cortical algorithm might be used in species that embody different sensory cells. Even more, any type of artificial sensor can be used as long there is an encoder (see 5.5.) that maps the sensor's output to the SDR.

Assume that the cortical space (brain) consists of $\mathbf{z}$ areas that implement the same or similar cortical algorithm described in this chapter. This area represents the *Cortical Unit* in this work. The Spatial Pooler algorithm described in the previous chapter assumes that the operating area consists of mini-columns defined by the area $C_i$ and their cell set $\mathbb{C}_{i.}$ The entire cell set across all areas can be generalized as the union of cell sets $\mathbb{C}_i$ of $\mathbf{z}$ areas.

$$\mathbb{C} = \bigcup_{\boldsymbol{i}=\mathbf{1}}^{\mathbf{z}} \mathbb{C}_i \mid z \in \mathrm{N} \tag{29}$$

As defined by equations (9) and (26), the spatial pooler function maps the encoded spatial input into the set of active mini-columns $C^a$ inside of a single operating area, let's say area $C_y$ .

$$sp\colon \boldsymbol{I} \rightarrow C^a \mid C^a \in C_y \tag{30}$$

In contrast to the SP, which uses mini-columns to encode the SDR, the building of associations uses cells and dendrite segments based on the model illustrated in Figure 10 and Figure 21. The neural association function ϕ takes a further step after the encoding has been completed in all areas to the set of active mini-columns or active cells (depending on the encoding algorithm). Based on rules described later in this section, the neural association algorithm activates a sparse set of cells $\mathbb{C}_y^a$ in the operating area $C_y$ in every learning cycle.

$$\phi\colon \mathbb{C}^{a} \rightarrow \mathbb{C}_{y}^{a} \mid \mathbb{C}^{a} \subset \mathbb{C}, \mathbb{C}_{y}^{a} \in C_{y}, \mathbb{C}_{y}^{a} \subset \mathbb{C}^{a} \qquad (31)$$

It maps the set of active cells $\mathbb{C}^{a} \subset \mathbb{C}$ into the new set of active cells $\mathbb{C}_{y}^{a}$. The association function $\phi$ uses as input the sparse set of a cell population from any area $\mathbb{C}_{i} \in \mathbb{C}$ in the given cortical space. Given the total T number of cells per mini-column, the entire cell set of cells of the area $C_{y}$ is defined as:

$$\mathbb{C}_{y} = \{c_{11}, .. c_{uw}, .., c_{MT}\}^{1\mathrm{x}(\mathrm{MxT})} \mid \mathrm{u} \in \{1, M\}, \mathrm{w} \in \{1, T\} \qquad (32)$$

The cardinality of $\mathbb{C}_{y}$ in the area is the total number of cells $\mathrm{MxT}$ across all mini-columns in the area $C_{y}$. Further, the cell and segment models are defined as tuples using the following equation.

$$\begin{gathered} cell_{ki} = \left\{S^{ki\rightarrow} = \{s_{ki1}, ., s_{kiE_{ki}}\}, \mathcal{E}^{ki} = \{\mathcal{E}_{ki1}, .., \mathcal{E}_{kiD_{ki}}\}\right\}, \\ \mathcal{E}_{kij} = \left\{cell_{ki}, S^{kij\leftarrow} = \left\{s_{uw1}, .., s_{kiR_{kij}}\right\}\right\} \mid \\ k \in \{1, M\}, i \in \{1, T\}, u \in \{1, M\}, w \in \{1, T\} \end{gathered} \qquad (33)$$

The model of the cell $i$ in mini-column $k$ is a tuple defined by the set of "outgoing" potential or connected synapses $S^{ki\rightarrow}$ from the cell and the set of cell segments $\mathcal{E}^{ki}$. Outgoing synapses connect the $cell_{ki}$ to segments of some other cells, which can belong to any area. Similarly, other cells from any other area might be connected to the cell $cell_{ki}$ via segment $\mathcal{E}^{ki}$. The number M specifies the total number of mini-columns in the area. The total number of segments of a cell change over time, and it is terminated by the value $D_{ki}$. In this context, the symbol '→' represents an outgoing synaptic connection from the cell, known as a pre-synaptic cell. Similarly, the symbol '←' represents the incoming synaptic connection to the segment. The index $i$ indicates one of the T cells inside of a mini-column.

This work assumes that every mini-column in the area contains the same number of cells. This can be easily changed in the future if required.

Every synapse has an associated permanence value (synaptic strength). Therefore, the set of (outgoing) synapses $S^{ki\rightarrow}$ contains receptor synapses with their permanence values $p_{ki}$. Receptor synapses connect pre-synaptic neurons (cells) with index $ki$ by their axons to the dendrite segment of some other neuron (postsynaptic cell). Due to structural plasticity, every cell has a different number of outgoing synapses, which is defined by the value $E_{ki}$.

The second set $\mathcal{E}^{ki}$ in the equation contains $D_{ki}$ dendrite segments (apical, distal or proximal) owned by the cell $cell_{ki}$, which is known as the parent cell of the segment $\mathcal{E}_{ki}$. Following the structural plasticity rules previously described (see 3.1), every cell can have a different number of dendrite segments, which are created and destroyed during the learning process.

Finally, the set of segments $\mathcal{E}^{ki}$ owned by the cell $cell_{ki}$ holds the set of (incoming) potential or connected $R_{kij}$ synapses $s_{uw}^{kij\leftarrow}$ sourced from other pre-synaptic cells $cell_{uw}$ (noted by symbol '$\leftarrow$') from an area. These synapses build connections to the segment $\mathcal{E}^{kij}$ whose owner cell $cell_{ki}$ is the postsynaptic cell of the cell $cell_{uw}$.

The cell in the segment definition and the segment set in the cell definition build a circular reference in the cell-segment model.

Note that the set of outgoing synapses of the cell $S^{ki\rightarrow}$ is defined by two indexes, mini-column index *k* and cell index *i*. The set of incoming synapses $S^{kij\leftarrow}$ is defined by three indexes. The cell is defined by mini-column index *k*, cell index *i* and the segment index *j*. The segment holds typically multiple synapses.

Outgoing synapse $s_{kij}$ from the set $S^{ki\rightarrow}$ can also be represented with the following equivalent expressions:

$$s_{kij} \triangleq s_{kij}^{uwz\rightarrow} \triangleq s_{ki}^{uwz\rightarrow} \qquad (34)$$

The synapse $s_{kij}$ belongs to the mini-column *k*, the cell $i$ and has an index j, where $0 < j < E_{ki}, j \in \mathcal{N}$. The equivalent notation of the same synapse $s_{kij}^{uwz\rightarrow}$ might sometimes be useful. This notation points out that cell $i$ in the mini-column *k* forms a synaptic connection *j* to the segment $\mathcal{E}_{uwz}$. The synapse index *j* is sometimes omitted $s_{ki}^{uwz\rightarrow}$, which indicates the connection of the outgoing connection from the $cell_{ki}$ to the segment $\mathcal{E}_{uwz}$. Similarly, incoming connections can be represented as $s_{uwl}^{kij\leftarrow}$ or $s_{uw}^{kij\leftarrow}$.

Please note that synaptic connections $s_{uwl}^{kij\leftarrow}$ and $s_{kij}^{uwl\rightarrow}$ are physically the same but represent a different point of view. $s_{kij}^{uwl\rightarrow}$ represents the connection of the synapse $l$ from the cell $cell_{uw}$ to the segment $j$ of the cell $cell_{ki}$. Similarly, $s_{uwl}^{kij\leftarrow}$ represents the connection $j$ to the cell $cell_{ki}$ from the synapse $l$ of the cell $cell_{uw}$ Because they are the same, they share the same permanence value $p_{uwl}^{kij} \triangleq p_{kij}^{uwl}$.

### 5.8.2 Learning Associations

If a dendrite segment is apical, distal or proximal, it can have two states in a learning iteration: matching or active.

The segment is, by default, the active one if the total count of connected synapses is greater than the segment *Activation Threshold* $\theta_a^{act}$, $\theta_d^{act}$ and $\theta_p^{act}$ respectively, for apical, distal (basal) or proximal segments (see 3.1.2). The segment is by default, the matching one if the total count of connected synapses on the segment is larger than the *Segment Minimum Threshold* $\theta_a^{min}$, $\theta_d^{min}$ and $\theta_p^{min}$ (respectively for the segment type) but less than a corresponding segment *Activation Threshold* $\theta_a^{axt}$, $\theta_d^{act}$ and $\theta_p^{act}$. The following equations define the set of active and matching distal segments.

$$\mathcal{E}_d^{act} = \{ \mathcal{E}_{kij}, .. \};\ \left(\sum p_{kij}^{uwl}\right) \geq \theta_d^{act}\ ; p_{kij}^{uwl} \quad > \theta_c \qquad (35)$$

The set of active distal segments $\mathcal{E}_d^{act}$is defined as a set of segments whose sum of connected synapses $s_{kij}^{uwl}$ with permanences $p_{kij}^{uwl} > \theta_c$ exceeds the segment *Activation Threshold* $\theta_d^{act}$. Similarly, the matching segment is defined as a set of segments that have a sum of potential synapses more significant than the *Segment Minimum Threshold* $\theta_d^{min}$.

$$\mathcal{E}_d^{match} = \{ \mathcal{E}_{kij}, .. \};\ \theta_d^{min} \leq \left(\sum p_{kij}^{uwl}\right) < \theta_d^{act}\ , p_{kij}^{uwl} > 0 \qquad (36)$$

The association between neurons and a segment is defined as a function that associates a set of neurons with a dendrite segment. The function *associate* (see Algorithm 9) is the lowest-level function in the neural association algorithm.

associate: $\mathbb{C}_x^a \rightarrow \mathcal{E}_{kij}\,, cell_{ki} \mid cell_{ki} \in\ C_y, \mathbb{C}_x^a\ \subset\ \mathbb{C}_x$ (37)

The function implicitly builds the N:1 relation between $\mathbb{C}_x^a$ a subset of active cells of the population $\mathbb{C}_x$ and the cell $cell_{ki}$ from the mini-column $C_y$, which is the owner of the associating segment $\mathcal{E}_{kij}$. The associating population of neurons is illustrated in Figure 28. Mini-columns with index $i$ and $j$ belong to the area $x$ and contain currently active cells with indexes $i1, i2\ and\ j2$. All cells in mini-columns $i$ and $j$ build synaptic connections to the cell *u2*. In this example, the mini-column $u$ belongs to a different area than mini-columns $i$ and *j*. This example shows that associating neurons do not have to belong to the same area.

The set of active cells (green) $\mathbb{C}_x^a = \{i1, i2, j2\}$ in the current learning cycle should be associated with the segment $\mathcal{E}_{u21}$ of the mini-column $u$ and the cell $u2$. Because outgoing synapses from cells $s_{i1}^{u21\rightarrow}, s_{i2}^{u21\rightarrow}, s_{j2}^{u21\rightarrow}$ from cells $i1, i2, j2$, respectively, to the segment $\mathcal{E}_{u21}$, are already connected, they will be strengthened. In contrast, synapses $s_{j1}^{u21\rightarrow}, s_{j3}^{u21\rightarrow}, s_{i3}^{u21\rightarrow}$ that also build connections to the segment and originate from cells *j1, j3* and i3, which are inactive in the current learning cycle (grey), will be weakened[1]. With this rule, the more often the cell is associated with the segment, the stronger the association between the cell and the segment will be.

By following the Hebb Rules described in 2.1 and 3, if incoming synapses $s_{uw}^{kij\leftarrow}$ of the dendrite segment $\mathcal{E}_{kij}$ located in the area *y* already form connections from a set of active cells $\mathbb{C}_x^a$ located in the area *x* defined by the set of outgoing synapses $S^{uw\ act\rightarrow}$, then their permanence values will be incremented with the value *permInc* (Lines 07 and 08) in Algorithm 9.
These synapses are calculated as the intersection between the set of outgoing synapses of active cells and the synapses at the dendrite segment $S^{kij\leftarrow} \cap S^{uw\ act\rightarrow}$.

[1] Permanences are only weakened during encoding over mini-columns in the Spatial Pooler (SP). If the area does not use a mini-column structure (all cells belong to the area without mini-columns), then the weakening of permanence values does not occur if cells are not active in the current cycle. In such areas, weakening is achieved through Long-Term Depression when a synapse remains inactive for an extended period.

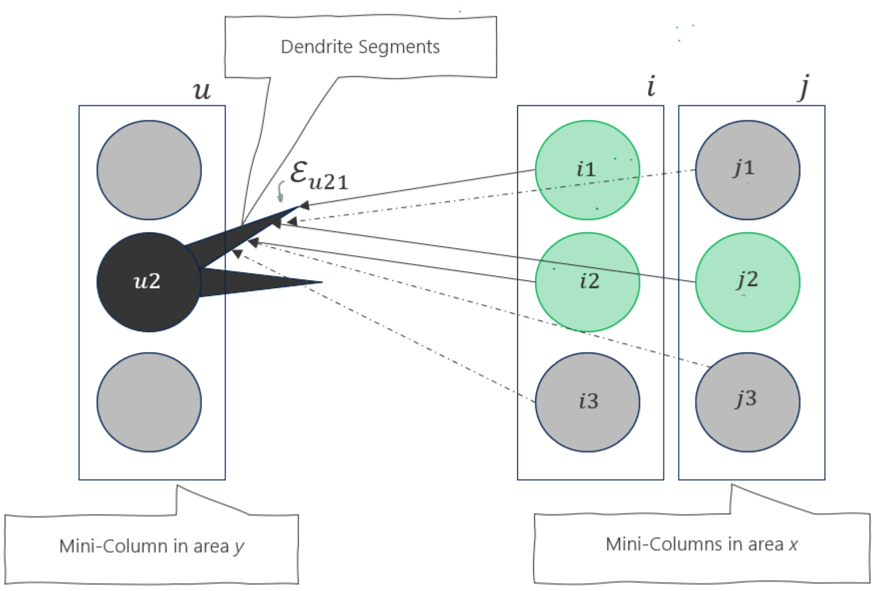


*Figure 28*

*Synaptic connections between pre-synaptic cells from area X to segments in area Y Cell i3 forms previously created synaptic connections. Cells i1, i2 and j2 will create new synaptic connections.*

In the example in Figure 28, the permanence value of synapses from cells i1, i2, and j2 will be incremented.

Permanences of all other synapses of the segment $\mathcal{E}_{kij}$, which are not connected with the synapses of cells $C^x$ calculated by expression $S^{kij\leftarrow} \cap S^{uw\,act\rightarrow} \in \emptyset$ will be decremented[1] with the value *permDec* (Lines 07 and 08). The association is remembered in the weight of the synaptic permanence values. In the example in Figure 28, the permanence value of synapses from cells i3, j1 and j3 will be decremented.

Increasing the synaptic permanence value strengthens the association between active pre-synaptic cells in area *x* and the postsynaptic cells in area *y*.
Similarly, decreasing the permanence value stimulates the forgetting of the association.

Finally, if the total count of connected synapses (cells) $R_{kij}$ is zero, the segment is destroyed (line 16).

Algorithm 9 Cell to Segment association

```
01| function associate ($\mathcal{E}_{kij}, \bigcup_z \mathbb{C}_i^{act}, permInc, permDec$)
02|    // $\mathcal{E}_{kij}$ : The segment in the area $\mathbb{C}_y$ to be associated with cells in area x.
03|    // $\bigcup_z \mathbb{C}_i^{act}$ : Set of cells from all areas, including area $\mathbb{C}_y$ to be associated with the
04|                   segment in the area $\mathbb{C}_y$.
05|    // $permInc$ : Permanence increment value when learning.
06|    // $permDec$: Permanence decrement value when forgetting.
07|    //Increment permanence of all synapses of cells in $\mathbb{C}^x$ connect to the segment $\mathcal{E}_{kij}$
08|    $p_{uwl}^{ki} \leftarrow p_{uwl}^{ki} + \delta \left\{ \begin{matrix} \delta = permInc; & S^{kij\leftarrow} \cap S^{uw\,act\rightarrow} \notin \emptyset \\ \delta = -1\ (or\ 0) * permDec & S^{kij\leftarrow} \cap S^{uw\,act\rightarrow} \in \emptyset \end{matrix} \right\} |$
09|                              $cell_{uw} \in \bigcup_z \mathbb{C}_i^{act},\ cell_{ki} \in \mathbb{C}_y$
10|    // $\delta = 0$ if the area is not mini-column based[1].
11|    // Destroy synapse if permanence is too small. Structural plasticity.
12|    IF $p_{uwl}^{ki} < 0.00001$
13|       $destroySynapse(p_{uwl}^{ki})$
14|    END
15|    // Destroy the segment if no cell is connected. Structural plasticity.
16|    IF $R_{kij} = 0$
17|       $destroySegment(\mathcal{E}_{kij})$
18|    END
19| end
```

## 5.8.3 Modelling structural plasticity

As previously discussed, structural plasticity is one of the essential findings that need to be supported in a cortical algorithm. The NAA defines this plasticity as a dynamic process of creating and destroying segments and synapses at the segment. In the established learning process, when synapses are already created between pre-synaptic cells, associations are created by Algorithm 9. However, learning is a dynamic process that not only manipulates synaptic weights (permanence values). The learning continuously enforces the creation of synapses, which is the opposite of destroying synapses in Algorithm 9, line 13. Synapses are created between pre-synaptic cells and a segment. Presynaptic cells can belong to the same or a different area than the destination (postsynaptic) cell, which owns the segment. Figure 29 shows synaptic connections between the segment of the cell in mini-column $u$ nested in the area $y$ and active pre-synaptic cells from mini-columns $i$ and $j$ nested in the area $x$.

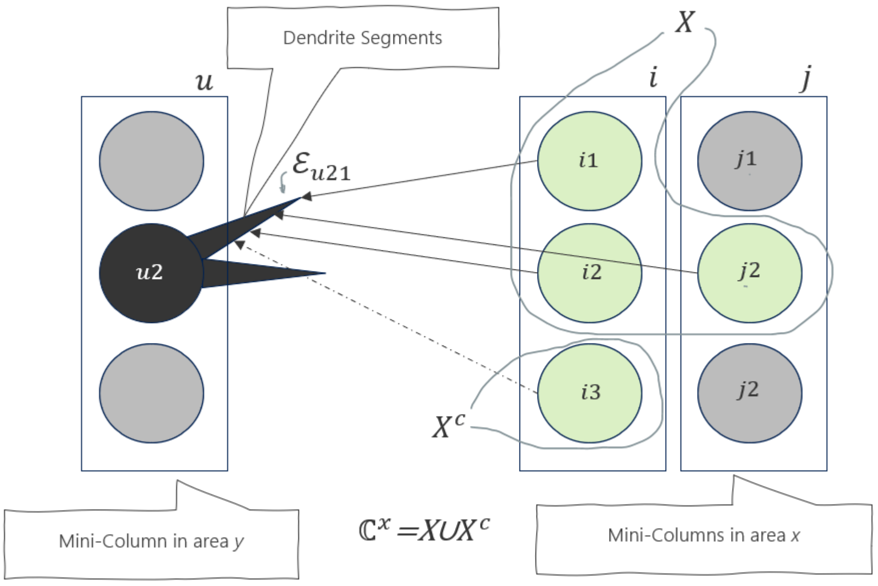


*Figure 29 Synaptic connections between pre-synaptic cells from the area $x$ to segments in the area $y$. Cell i3 forms previously created synaptic connections. Cells i1,i2 and j2 will create new synaptic connections.*

Assume active cells *i1, i2* and *j2* do not form synapses to the segment *u21*. In contrast, the active cell i3 is already synaptically connected to the segment *u21*. Structural plasticity runs two algorithms in this context. The first one, Algorithm 10, creates new synaptic connections between the population of neurons shown in Figure 29. The second one, Algorithm 11, is responsible for creating segments.

Given the set of associating cells $\mathbb{C}_x^a$, the segment *j* of the mini-column *k* and the cell *i* the function *createSynapses* in Algorithm 10 creates *cnt* number of new (incoming) synapses $S^{kij\leftarrow}$ at the segment $\mathcal{E}_{kij}$. In contrast to Algorithm 9, which performs learning (strengthening and weakening) on existing synaptic connections in Figure 29, Algorithm 10 forms new synaptic connections and implements structural plasticity (see 2.6).
Instead of associating active cells $\mathbb{C}_x^a$ from the single area $x$ as shown in Figure 29, the algorithm operates rather on the union set $\bigcup_z \mathbb{C}_i^{act}$of active cells from Z associating areas. First, the set $X$of cells in $\bigcup_z \mathbb{C}_i^{act}$ that already forms synapses with the segment $\mathcal{E}_{kij}$ is calculated (line 09). The complement set $X^c$of $X$ holds all remaining cells in $\bigcup_z \mathbb{C}_i^{act}$ that do not form synapses to the segment (Line 11).

Algorithm 10 Structural plasticity with synapse creation

```
01| function createSynapses ( $\mathcal{E}_{kij}, \bigcup_z \mathbb{C}^{i-act}, maxs$)
02|   // $\mathcal{E}_{kij}$: The Segment to create synapses that will connect presynaptic cells.
03|   // $\bigcup_z \mathbb{C}^{i-act}$: List Presynaptic active cells from different areas, excluding $\mathbb{C}_y$
04|   //           that will be connected to the segment with a new synapse.
05|   //           In a case of bursting, these are winner cells.
06|   // $maxs$: The required number of synapses to be created.
07|   // $p_0$: Initial synaptic permanence value
08|   // X is a set of cells that already have a synaptic connection to the segment.
09|   $X = \{ cell_{uw} \in \bigcup_z \mathbb{C}^{i-act};\ S^{kij\leftarrow} \cap S^{uw\rightarrow} \notin \emptyset \}$
10|       // Create a required number of synapses, from non-connected active cells
11|       FOR $i = 1, i < \min(maxs, |X^c|; \bigcup_z \mathbb{C}^{i-act} = X \cup X^c)$
12|          // Get random cell from cells in $\mathbb{C}^x$ that do not create the synapse to segment.
13|          $cell_{uw} \leftarrow random(X^c) \mid u \in \{1, M\}, w \in \{1, T\},\ cell_{uw} \notin X^c$
14|          // createSynapse from the cell to the segment
15|          createSynapse( $cell_{uw},\ \mathcal{E}_{kij},\ p_0$) | $cell_{uw} \notin X^c$
16|       ENDFOR
17| end
```

The loop in lines 11-16 randomly selects cells from $X^c$and create the synapse from the cell to the segment until *cnt* synapses are created or all cells in $\bigcup_z \mathbb{C}_i^{act}$ form outgoing synapses $S^{uw\rightarrow}$to the segment $\mathcal{E}_{kij}$.

At the beginning of the Association Algorithm, no synaptic connections exist in the population of neurons: $S^{ki\leftarrow} = \emptyset, S^{uw\rightarrow} = \emptyset$. Connections are created and destroyed dynamically by using Algorithm 9 and Algorithm 10.

Another important part of structural plasticity is the bursting algorithm. The bursting of the active-mini mini-column (see Algorithm 11) is a process divided into two steps: Learning Associations at the winner segment (Lines 10-18) and Creation of new segments (Lines 19-27).

Both steps are responsible for creating segments at cells, creating and growing synapses at segments and learning associations (strengthening or weakening synapses) at the segment.

Algorithm 11 Structural plasticity with column bursting

```
function burstMiniColumn ($c_k, \bigcup_z \mathbb{C}_i^{act}, \mathbb{C}^{win}, permInc, permDec, maxS$)
  // $c_k$ : The bursting mini-column.; $c_k \in C_y$
  // $\mathbb{C}^{win}$ : Presynaptic winner cells from the last iteration.
  // $\bigcup_z \mathbb{C}_i^{act}$List of sets of associating active cells.
  // $permInc$ : Permanence increment value when learning.
  // $permDec$: Permanence decrement value when forgetting.
  // $maxS$: Maximal number of allowed new synapses at the segment.
  //Form the set of matching segments in the mini-column.
  $E \leftarrow \{\mathcal{E}_i^{match}$ in $c_k\}; \theta_d^{min} \le (\sum_i \delta; \begin{cases} \delta = 1; p_i \ge 0 \\ \delta = 0; p_i < 0 \end{cases}) < \theta_d^{act}$
  IF $E \notin \emptyset$ // If there are matching segments.
      // Find the segment in a mini-column with the maximal number
      // of potential synapses. Calculate a maximal number of potential synapses.
      $\mathcal{E}^{max} \leftarrow \max\{|S^{kij\leftarrow}|\} \mid S^{kij\leftarrow} \in \mathcal{E}_{kij},\ \mathcal{E}_{kij} \in E$
      // associate the segment.
      associate($\mathcal{E}^{max}, \bigcup_z \mathbb{C}_i^{act}, permInc, permDec$)
      // Create a set of new synapses.
      createSynapses($\mathcal{E}^{max}, \mathbb{C}^{win}, maxS - m) \Leftrightarrow maxS - m > 0$
      return ({ $c_{ki}$ }, $cell^{max}$| $i \in \{1,2,..T\}$)
  ELSE // If there are no matching segments
        // Find the cell with the least number of segments in the mini-column k.
        ($num, cell^{k-min}$ ) $\leftarrow min\{|\mathcal{E}_{ki}|\} \mid \mathcal{E}_{ki} = \{\mathcal{E}_{kij}\}, 1 < \mathrm{i} \le \mathrm{T}, 1 < \mathrm{j} \le D_{ki}$
        // Create the new segment.
       $newSeg \leftarrow$createSegment($cell^{k-min}$)
      // Create a set of new synapses at the new segment.
      createSynapses($newSeg, \mathbb{C}^{win}, maxS - |\mathbb{C}^{xwin}|$)
      return ({ $c_{ki}$ }, $cell^{k-min}$| $i \in \{1,2,..T\}$)
  ENDIF
end
```

The bursting of the mini-column is initiated if the mini-column has no active segments. The mini-column bursts (activates) all cells in the mini-column if the mini-column has no active segments. Some papers related to HTM theory define bursting at the cell level (see 3.1), which differs from mini-column bursting described in this section. The goal of bursting is to form new synapses that activate segments. According to equation (36) segment is considered as matching one if the total count of connected synapses (see permanence connected threshold $\theta_c$ *in* section 5.3) at the segment reaches thresholds $\theta_d^{min}$. The bursting algorithm distinguishes between mini-columns that have a number of segments greater than $\theta_d^{min}$ (matching-segments threshold) and lower than $\theta_d^{act}$

(active segment threshold). At the beginning (Line 09), the set of matching segments $E$ of the processing mini-column is calculated.

### Step 1: Learning associations at the matching segment with the most synapses

If the mini-column <u>has</u> matching segments and no active segments (Line 10), the segment $\mathcal{E}^{max}$ with the maximum number of potential synapses (see 5.2) is calculated in Line 13. Then, that segment is chosen as a "*winner*" for learning associations (Line 15). Learning associations will strengthen or weaken synapses (see Algorithm 9). As next (Line 17), more synapses are created at the "*winner*" segment up to the required number of synapses $maxS$. All other non-"*winner*" segments in the mini-column are ignored in the current turn. At the end (Line 18), the algorithm returns all cells as active cells (*bursting*) and selects the cell $cell^{max}$ as the *winner*-cell (best one) with the highest number of synapses. This step makes sure that the association between a set of active cells $\mathbb{C}^{xact}$ and the best *winner*-segment with the highest number of synapses is strengthened, which will, over time, build a sparse contextual dependency (association) between a context encoded by a sparse set of active cells $\mathbb{C}^{xact}$ and the *winner*-segment.

### Step 2: Creation of a new segment if not matching and active segments exist

If the mini-column <u>has no</u> matching segments (the number of synapses of all segments is less than $\theta_d^{min}$) the algorithm will initiate creating a new segment (Line 19). In this case, the least used cell has to be found (Line 21) in the active mini-column $c_k$. That is the cell with a minimum number of segments. For every cell $cell_{ki}$ (Line 21) in the mini-column, the set of segments $\mathcal{E}_{ki} = \{\mathcal{E}_{kij}\}$ is looked up and the $cell^{k-min}$ with the minimum number of segments $min\{|\mathcal{E}_{ki}|\}$ is selected as the least used cell. Finally, a new segment is created for the $cell^{k-min}$ (Line 23). At the end of this step, all cells in the mini-column are returned as active cells. The cell $cell^{k-min}$ with the minimum segments is chosen as the winner one.

### Summary

To recap, in both steps bursting the mini-column activates all cells in the mini-column if the mini-column has no active segments. However, the rule for choosing the *winner* cell is different. In the case of matching segments (step 1), the cell owner of the segment with the highest number of potential synapses is chosen as the "*winner*" cell.

In step 2, a new segment is created in the cell with the lowest number of segments. That cell is marked as the *winner* cell, and it will be preferred for growing synapses in the next learning cycle.
Outgoing synapses of all active cells in the cycle are strengtheneth or weakened (see Algorithm 9 and Algorithm 11).
Finally, new synapses are created on the *winner* cells only.

### 5.8.4 Associating by cellular activation

As defined by equation (31), the cellular association Algorithm 12 activates a sparse set of cells in the target area that belong to mini-columns that form the area *y* (see Figure 28). Given the operating area $y$ with the set of mini-columns $C_y = \{\, c_1,\, c_1, .., \, c_M\}$, and sets of active segments $\mathcal{E}_d^{act}$ and matching segments $\mathcal{E}_d^{match}$ of any kind (*a, d* or *p* - see 5.8.2) the cellular activation algorithm traverses all of the mini-columns in the area *y* in every learning iteration and activates cells in $C_y$ according to equation (31). Note that this algorithm operates on mini-columns that hold cells of the specific area $C_y$. The NAA does not propose how mini-columns should be activated. This work focuses the encoding by Spatial Pooler, which encodes the spatial pattern into a set of active mini-columns. However, some future versions of NAA might use an additional encoding mechanism like grid cells, which would encode the location information with the *Location Context* rather than spatial information with the *Spatial Context*.

In generalized form, the cellular activation takes as input the active cells from the entire cell population $\mathbb{C}$ defined by equation (29), but it activates only cells in its operating area $y$ of the population $\mathbb{C}_y$. Associating active cells from specific area to active cells of some other or the same area creates meaning in the given context. The set $\bigcup_z \mathbb{C}_i^{act}$ is represented as a union or list of sets of associating active cells that belong to multiple areas. Additionally, the algorithm uses also as input the set of winner cells $\mathbb{C}^{ywin}$ from operating area $y$. Winner cells are cells chosen for growing of synapses in the current cycle according to rules of structural plasticity as described later in this section. The result of this algorithm in every learning cycle is an updated set of chosen active $\mathbb{C}^{yact}$ and winner $\mathbb{C}^{ywin}$ cells.

Algorithm 12 performs the following tasks:

1. <u>Weak existing associations at matching segments of inactive mini-columns</u> (Lines 11-19)

   For all mini-columns that are not active in the current cycle but hold cells with matching segments as defined by equation (36), the permanence value of synapses $S^{uw\rightarrow}$ from presynaptic cells $\bigcup_z \mathbb{C}_i^{act}$ to the matching segments of the inactive mini-column are decremented by the value *permDec* (Lines 13-18). One of the major principles of this algorithm is that all learning operations should happen on cells and synapses inside active mini-columns only. Therefore, in this step, no active and winner cells are selected.

2. <u>Strengthening the existing associations in active mini-column</u> (Lines 20-29)

   a. Strengthening the existing associations of active segments inside active mini-columns. If the mini-column is the active one and has active segments as defined by equation (35), then existing synapses $S^{uw\rightarrow}$ from presynaptic cells $\bigcup_z \mathbb{C}_i^{act}$ to the active segment are strengthened. (Lines 20-28).

   b. In the same turn, the number of synapses at active segments of the active mini-column is grown. Active segments will create new synapses (Line 26) to the specified limit. The algorithm specifies the maximal number of synapses at the segment *maxS* (Line 07). Having the set of synapses $S^{act\leftarrow}$ at the segment $\mathcal{E}^{act}$ according to equation (32), the remaining (missing) number of synapses at the segment is calculated from the required number of potential synapses $maxS$ and the number $|S^{act\leftarrow}|$ of potential and connected synapses at the active segment $\mathcal{E}^{act}$.

   c. In Line 28, cells that are owners of the active segment $\mathcal{E}^{act}$ are chosen as active and winner cells in the current iteration (see equation (35).

   This step completes with sets of active and winner cells.

3. Column Bursting if no segment is active at the active mini-column
   (Lines 30-33)

   Cells with no active segments create new segments and form new synapses. This process is called *bursting,* as described in the previous section. The bursting process results in a set of active cells in the operating area that contain all mini-column cells from that area. In this step, the single winner cell is chosen. This is the cell $cell^{k-min}$ with the minimum number of segments (see Algorithm 11, Lines 20-26).

The cellular activation algorithm executes three major tasks. First, it weakens (punishes) existing associations at matching segments of inactive mini-columns. Second, it strengthens the existing associations between active cells in the entire population $\bigcup_z \mathbb{C}_i^{act}$and active segments in the active mini-column of the operating area $\mathbb{C}^y$. In the same turn, new synapses are created between winner cells in the area $\mathbb{C}^x$ and active segments in active mini-column of the area $\mathbb{C}^y$. Third, the active mini-column is burst if there are no active segments in the mini-column.

The result of the algorithm is an updated set of chosen active $\mathbb{C}^{yact}$ and winner $\mathbb{C}^{ywin}$ cells.

Algorithm 12 Cellular Activation

```
function activateCells ($\bigcup_z \mathbb{C}_i^{act}$ , $\mathbb{C}^{ywin}$, $permDec$, permInc, $maxS$, $C_y$)
  // $\bigcup_z \mathbb{C}_i^{act}$: List of sets of active cells from all areas including area y.
  // $\mathbb{C}^{ywin}$: Winner cells from mini-columns space the area $C_y$.
  // $permInc, permDec$: The perm. Increment/decrement when learning.
  // $maxS$: Maximal number of new synapses at the segment.
  // $C_y$: The area with mini-columns that hold cells $\mathbb{C}_i^{act}$, $\mathbb{C}^{ywin}$

  // Iterates all mini-columns in the area
  FOREACH $c_k$ IN $C_y$
          // If the mini-column is NOT active in the current iteration.
          IF $c_k \notin C^a$
              // Punish inactive mini-column.
             FOREACH $\mathcal{E}^{matching}$ IN $c_k$
                   // Gets the set of presynaptic cells of the matching segment.
                   $\mathbb{C}^{\rightarrow} = \{ cell_{uw} \} | S^{uw\rightarrow} \in \mathcal{E}^{matching}, cell_{uw} \in \bigcup_z \mathbb{C}_i^{act}$
                   // Decrements existing synapses on the segment.
                   associate($\mathcal{E}^{matching}$, $\mathbb{C}^{\rightarrow}$, $-1 * permDec$, 0)
              ENDFOR
           ELSE If the mini-column is active in the current iteration.
              FOREACH $\mathcal{E}^{act}$ IN $c_k$ // Process active segments in the active mini-column.
                   // Gets the set of presynaptic cells of the active segment.
                   $\mathbb{C}^{\rightarrow} = \{ cell_{uw} \} | S^{uw\rightarrow} \in \mathcal{E}^{act}, cell_{uw} \in \bigcup_z \mathbb{C}_i^{act}$
                   // Increment existing synapses on the segment.
                   associate($\mathcal{E}^{act}$, $\mathbb{C}^{\rightarrow}$, $permInc$, $permDec$)
                   // Increment number of synapses at the segment.
                   createSynapses($\mathcal{E}^{act}$, $\mathbb{C}^{\rightarrow}$, $maxS - |S^{act\leftarrow}|$)
                   // Increment number of synapses at the segment.
                   $\mathbb{C}^{ywin} = \mathbb{C}^{yact} \leftarrow \bigcup_{i=1}^{|S^{act\leftarrow}|}(c_i) \mid s_i^{\leftarrow} \in \mathcal{E}^{act}$
               ENDFOR
               $E \leftarrow \{\mathcal{E}^{act}$ in $c_k\}$ // Get the set of active segments.
               IF $|E^c| > 0$ // Process non-active segments at the mini-column.
                  $(\mathbb{C}, c) \leftarrow$ burstMiniColumn( $c_k$, $\bigcup_z \mathbb{C}_i^{act}$ , $\mathbb{C}^{ywin}$, $permInc$, $permDec$, $maxS$)
                  $\mathbb{C}^{ywin} \leftarrow c$, $\mathbb{C}^{yact} \leftarrow \mathbb{C}$
               ENDIF
           ENDIF
   ENDFOR
end
```

### 5.8.5 Associating by segment activation

The previous section describes how the NAA activates cells using various biological rules. The result of the cellular activation algorithm is the population of active and *winner* cells $\mathbb{C}^{yact}$ and $\mathbb{C}^{ywin}$. Winner cells do not play any role in the activation of segments. The cellular activation executes at active and matching segments in the mini-column. Selecting segments into active and matching is part of the segment association algorithm described in this section.

The task of this part of the NAA is to activate segments for the next cycle $t_{+1}$. Segments activated in the current cycle $t_{+0}$ for the next cycle are used to activate cells in the next cycle $t_{+1}$ (see 5.8.4). Also, cells activated in the $t_{+1}$ are used to activate segments for the cycle $t_{+2}$.

The activation of segments starts with counting all synapses of every presynaptic cell in a set of active cells $\mathbb{C}^{yact}$ and looks up connected segments of their postsynaptic cells. It takes cells in $\mathbb{C}^{yact}$ as pre-synaptic cells and follows their outgoing synapses, which connect to destination segments as defined by (33). Destination cells can theoretically be a part of any area. However, the limitation of the current algorithm implementation assumes that connecting cells are in the same area.

The outgoing synaptic connections of active cells in $\mathbb{C}^{yact}$ keep memorized synaptic associations to some other cell population $\mathbb{C}^{zact2}$. In other words, the population $\mathbb{C}^{yact}$associates to population $\mathbb{C}^{zact2}$. The intensity of the association is calculated with synaptic activity in the function *countSynapses* of Algorithm 13 in Lines 17-32. This function finds all following segments connected by outgoing synapses of pre-synaptic cells in $\mathbb{C}^{yact}$ and calculates how many potential- and active synapses are at each connected segment in lines 27 and 29, respectively.

The segment activation starts with calculating the synaptic activity in line 10. The segment from the area $C^{y}$ is added to the list of active segments (Line 12) if the total count of connected synapses is over the threshold $\theta_d^{act}$ as defined by equation (35). Similarly, the segment from the area $C^{y}$ is added to the list of matching segments if the total count of potential synapses is over the defined threshold $\theta_d^{min}$ as defined by equation (36).

Algorithm 13 Segment Activation

```
function activateSegments ($\theta_c, \mathbb{C}^{xact}, \mathbb{C}^{yact}, \theta_d^{act}, \theta_d^{min}$)
  // $\theta_c$: Connected Permanence threshold
  // $\mathbb{C}^{yact}$: Set of active associating cells in area y
  // $\mathbb{C}^{xact}$: Set of active associating cells in area x
  // $\theta_d^{act}$: Number of required synapses to activate the segment
  // $\theta_d^{min}$: Number of required synapses to declare matching segment
  // Initialize empty lists for active and matching segments
  $actSegs[] = \emptyset, matchSegs[] = \emptyset$
  // calculate potential and connected synapse counters.
  $(synCntMap^{pot}, synCntMap^{act}) \leftarrow$ countSynapses($\mathbb{C}^{yact}, \theta_c$)
  // Add segment as active if number of connected (active) synapses overcomes $\theta_d^{act}$
  $\mathcal{E}^{yact} \leftarrow \mathcal{E}_{kij} \Leftrightarrow synCntMap^{act}[\mathcal{E}_{kij}] \geq \theta_d^{act}, \mathcal{E}_{kij} \in \mathrm{C}^y$
  // Add segment as matching if number of connected synapses overcomes $\theta_d^{min}$
  $\mathcal{E}^{ymatch}[] \leftarrow \mathcal{E}_{kij} \Leftrightarrow synCntMap^{match}[\mathcal{E}_{kij}] \geq \theta_d^{min}, \mathcal{E}_{kij} \in \mathrm{C}^y$
end

function countSynapses ($\mathbb{C}^{act}, \theta_c$)
  // $\mathbb{C}^{act}$: Set of activated (spiking) cells in area(s)
  // $\theta_c$: Connected Permanence threshold
  // Defines maps for segment synaptic counters.
  $synCntMap^{pot}[] = \emptyset, synCntMap^{act}[] = \emptyset$
  // Traverse all presynaptic active cells.
  FOREACH $cell$ IN $\mathbb{C}^{act}$
    // Gets the connecting segment of the presynaptic cell
    $\mathcal{E}_{kij} \leftarrow cell$
    // Increment counter of potential synapses to the segment $\mathcal{E}_{kij}$
    $synCntMap^{pot}[\mathcal{E}_{kij}] \mathrel{+}= \sum_{u,w} \delta \begin{Bmatrix} \delta = 1; \Leftrightarrow & p_{uw}^{kij} > 0 \\ \delta = 0; \Leftrightarrow & p_{uw}^{kij} = 0 \end{Bmatrix}$
    // Increment counter of active synapses to the segment $\mathcal{E}_{kij}$
    $synCntMap^{act}[\mathcal{E}_{kij}] \mathrel{+}= \sum_{u,w} \delta \begin{Bmatrix} \delta = 1; \Leftrightarrow & p_{uw}^{kij} \geq \theta_c \\ \delta = 0; 0 < & p_{uw}^{kij} < \theta_c \end{Bmatrix}$
   ENDFOR
  return ($synCntMap^{pot}, synCntMap^{act}$)
end
```

### 5.8.6 Learning Associations by cellular and segment activation

The previous sections in this chapter describe structural plasticity, cellular activation and segment activation. Learning associations is a process that associates a cell population from an area with the cell population of the same or another area. For example, colour receptor SDRs can be associated with location cells. Association learning is not only limited to sensory cells. They can be built between any cell population and any area.

To understand how associating works, assume the learning should be processed in the area $\mathbb{C}_y$. The same learning happens in every area separately synchronized by the clock that defines a learning cycle (iteration). The learning area $y$ is connected with another associated area $x$ (Figure 30). The area $y$ learns the temporal associations between cell populations inside $y$ and associations from the population in $X$. Temporal associations inside of $y$ are created between active cells in the cycle $\mathbb{C}^{yact}(t_n)$ and active cells in the cycle $\mathbb{C}^{yact}(t_{n+1})$ .

The learning algorithm is executed in the function *compute* in Algorithm 14. It is divided into two steps. In the first step (Line 08), the learning implements Algorithm 12, described in section 5.8.4, which chooses the set of active cells in the current iteration $t_n$ for the given input ($\bigcup_z \mathbb{C}_i^{act}$ , $\mathbb{C}^{ywin}$) based on the current segment state. Chosen cells are used (Line 10) to suggest active segments for the next cycle.

Algorithm 14 Neural Association computation

```
01| function compute($\bigcup_z \mathbb{C}_i^{act}$, $\mathbb{C}^{ywin}$, $permDec$, permInc, $maxS$, $C_y$)
02|   // $\bigcup_z \mathbb{C}_i^{act}$: List of sets of active cells from all areas. $\mathbb{C}^{yact} \in \bigcup_z \mathbb{C}_i^{act}$
03|   // $\mathbb{C}^{ywin}$: Winner cells from mini-columns space the area $C_y$.
04|   // $permInc$, $permDec$: The perm. Increment/decrement when learning.
05|   // $maxS$: Maximal number of new synapses at the segment.
06|   // $C_y$: The y area with mini-columns that hold cells $\mathbb{C}^{yact}$, $\mathbb{C}^{ywin}$
07|   // Activate new cells for the current learning iteration (cycle)
08|      ($\mathbb{C}^{ywin}$, $\mathbb{C}^{yact}$) ← activateCells($\bigcup_z \mathbb{C}_i^{act}$, $\mathbb{C}^{ywin}$, $permDec$, permInc, $maxS$)
09|   // Activate new segments for the current learning iteration (cycle)
10|    $\mathcal{E}^{yact}$, $\mathcal{E}^{ymatch}$ ← activateSegments($\theta_p$, $\mathbb{C}^{xact}$, $\mathbb{C}^{yact}$, $\theta_d^{act}$, $\theta_d^{in}$)
11|      RETURN ($\mathbb{C}^{yact}$, $\mathbb{C}^{ywin}$, $\mathcal{E}^{yact}$, $\mathcal{E}^{ymatch}$)
12| end
```

The selection of segments for the next cycle semantically represents the prediction capability of the NAA. The result of the execution of learning associations is a state, defined as a tuple of winner cells of the learning area, active cells of the learning area, matching segments and active segments of the learning area.

$$(\mathbb{C}^{yact}, \mathbb{C}^{ywin}, \mathcal{E}^{yact}, \mathcal{E}^{ymatch})(t_{n+1}) \leftarrow \\ (\mathbb{C}^{ywin}, \mathcal{E}^{yact}, \mathcal{E}^{ymatch}, \bigcup_{z} \mathbb{C}_i^{act})(t_n) \tag{38}$$

The process of learning in area y is illustrated in Figure 30. In the example, area y is a learning association from area x. In the iteration $t_n$. area x has two active cells (top left). In this context, it is not of interest how area x is built and how cells are activated. At the same time, area y has two active cells inside two active mini-columns. Mini-columns are activated as a result of the SP algorithm described in 5.7. They encode the proximal spatial input of the SP (bottom left). By activating mini-columns, Algorithm 12, starting in condition at Line 20, selects cells that will be activated. After the selection of active cells in area y, the Segment Activation (Algorithm 13) predicts segments that are expected to be active in the next iteration. Active cells in the iteration $t_n$, built outgoing synapses to some other cells by using Algorithm 12 and possibly some other areas (not illustrated in the figure). At the end of the iteration, the area y state is defined by two predicted active segments (Figure 30 - right) owned by two cells expected to be active in the next iteration. The next iteration $t_{n+1}$. starts in the area y again by activating mini-colums (shown in green frame at right) through some other proximal spatial input streamed in the SP. At the same time, some other cell populations (two cells at the top right) are activated in area x. Again, the cell activation algorithm (Algorithm 14, Line 08) starts to activate cells based on the segment state calculated in the iteration $t_n$ (Algorithm 14 Line 10). Synapses from all active cells to activated cells are strengthened and existing synapses from non-active cells (red) are weakened.

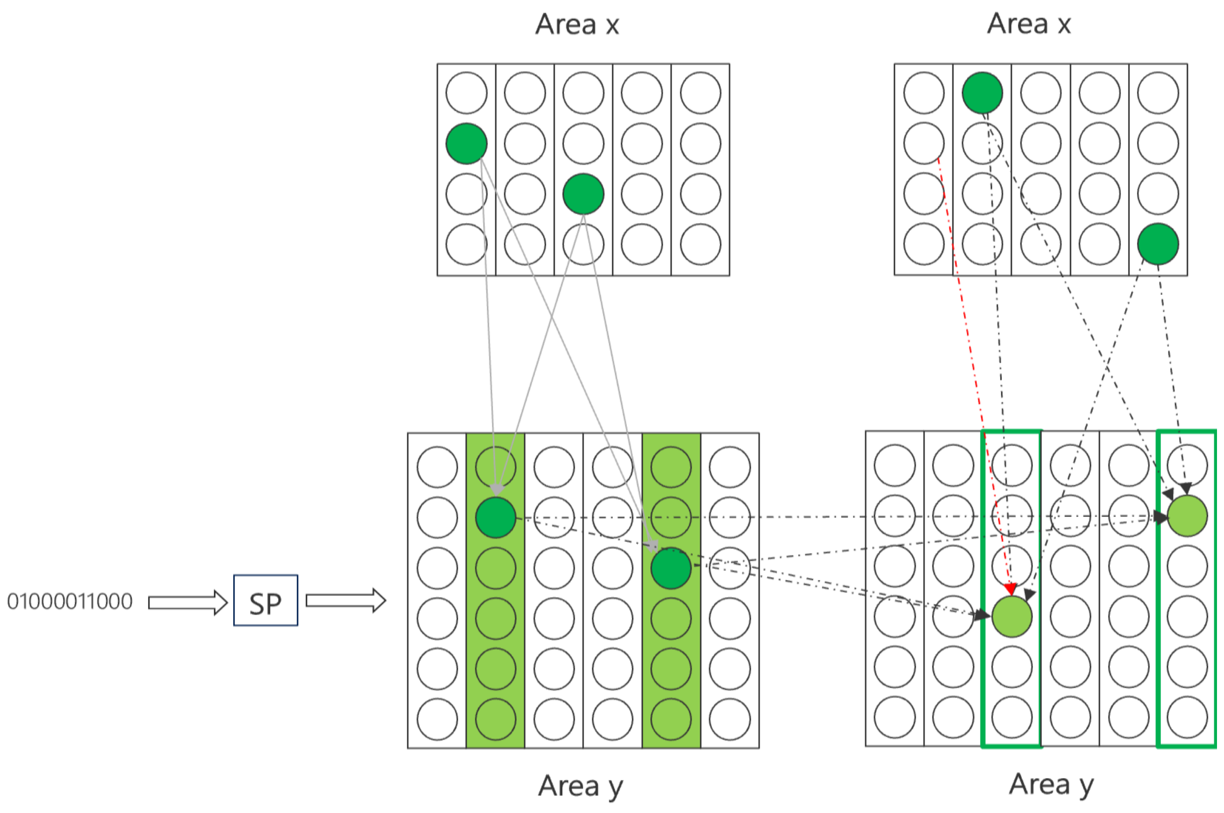


*Figure 30 Temporal association over multiple areas*

The learning association algorithm builds associations through synaptic connections, the Hebb rule, structural plasticity etc. It dynamically grows segments and synapses and can predict the next state. If the predicted state occurs, the permanence values of synapses are increased. Synapses between falsely predicted cells are punished, which is the main feature of reinforcement learning (see Chapter 1).

Aligned to biology described in Chapter 2, Neural Association Algorithm described in this chapter demonstrates how areas, mini-columns, synapses, segments and neural cells play together to build cognitive capabilities like associations and prediction. From the point of view of a learning area y in the last example, the proximal segment has the most significant impact on the activation of cells, as described in 2.5. By building associations between cells inside of the same area, the temporal association is created between the current state (set of active segments $\mathcal{E}^{yact}$ and cells $\mathbb{C}^{xact}$ and the next state. However, the next state is also influenced by synaptic connections from other areas. Semantically, connections from other areas define the context for associations. The algorithm, by default, continuously learns and also forgets, which is known as online learning. It learns the “world” presented by proximal spatial inputs from some sensory areas. Figure 31 illustrates the transition of the learned state in a simplified form. Every circle inside a box represents the set of active cells inside the area. Different position of the circle represents a different set of active cells. Figure 31 has the same meaning but a simplified representation of the cell state shown in Figure 30.

Area Y learns associations from area X that provides spatial contextual information to area Y. Additionally, area Y forms synapses inside the area. These synapses create another kind of temporal association. Every cell state in the area represents some encoded event or a spatial state. Associating (outgoing) synapses from active cells of the event state suggests (predicts) the next set of active cells that encode the following event and so on.

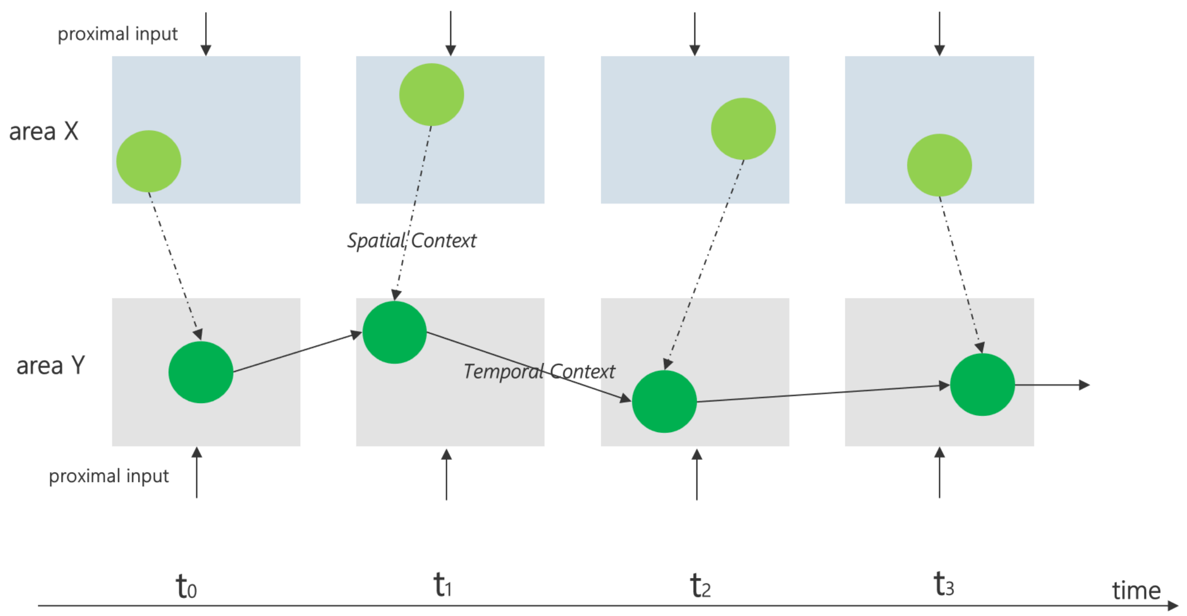


*Figure 31 Temporal Association Learning Path. Simplified representation*

The set of temporal associations in area Y forms a *temporal association path*. Such temporal association between multiple events (elements) can learn sequences. Every association consist of the spatial contextual information from another area and *Temporal Context* in its area. Theoretically, the association algorithm can form a spatial and temporal association between all areas.
Area Y has enough information to predict the next active state. Theoretically, area Y has enough information to follow the *temporal association path* to predicting the state in

any future iteration, assuming that the proximal spatial input does not change over time.

### 5.8.7 Summary

This section briefly describes the Neural Association Algorithm, which can create any kind of association like spatial and temporal associations. It aligns with the biological findings described in Chapter 2 and generalizes the idea of HTM described in Chapter 3. The algorithm operates on multiple cortical areas and hypothesizes that all internal states in every area are represented as SDR encoded in any possible way. During the learning process, NAA dynamically creates and destroys segments and synapses. As a result, synaptic connections between cells and segments in areas create associations between SDRs. Associations can be made between spatial states or can have a temporal character. Spatial associations demonstrate the capability of the algorithm to learn spatial patterns. Similarly, temporal associations illustrate the ability of NAA to learn sequences. Section 9.4 discusses how these capabilities can be used to create a context and a semantical meaning.

## 5.9 Design and Implementation of the Temporal Memory

Temporal memory, the next important algorithm in this research, belongs to the HTM concept described in 3.1.4. In contrast to SP, which learns spatial patterns, the Temporal Memory (TM) algorithm is responsible for learning sequences. The input of the TM consists of two parts. The first input includes the mini-column SDR encoded by the SP defined by equation (26). The second input is the set of active cells inside the learning area in the current iteration. This input represents the temporal context for the next iteration step. The output of TM is the set of active cells that belong to active mini-columns activated by the SP.

The Temporal Memory Algorithm is a special case of the Neural Association Algorithm. By removing associating area X and running the algorithm inside area Y, the set of associating cells is reduced to the set of active cells in area Y.

$$\bigcup_{z} \mathbb{C}_i^{act} = \mathbb{C}^{yact} \tag{39}$$

With this restriction, the Temporal Memory algorithm reduces the generalized Neural Association model defined by (38) to the following equation:

$$(\mathbb{C}^{yact}, \mathbb{C}^{ywin}, \mathcal{E}^{yact}, \mathcal{E}^{ymatch})(t_{n+1}) \leftarrow (\mathbb{C}^{ywin}, \mathbb{C}^{ywin}, \mathcal{E}^{yact}, \mathcal{E}^{ymatch})(t_n) \tag{40}$$

The only contextual information in TM is the previous state of active cells because no other areas are involved in the learning process. Because the only existing *Spatial Context* is the set of active cells in the same area, the same active cells in every learning cycle define a Temporal Context for the upcoming cycle semantically. In other words, cells that are active in cycle $t_n$, predict cells that will be active in the cycle $t_{n+1}$, by their associating synapses. With this, the TM learns sequences by creating temporal associations between a set of cells in the previous iteration with the set of cells in the current iteration.
Figure 32 shows a simple area with four mini-columns containing four cells each. The model in the figure comprises four mini-columns and assumes that spatial element 'A' activates the first mini-column, the element 'B' the second one etc. Every mini-column contains four cells. This model is unrealistic because the number of mini-columns and cells is too low to build a usable SDR. This simplified model is chosen to illustrate the temporal learning process. The SDR in this model will activate a single mini-column, which is simplified, and not a realistic case.

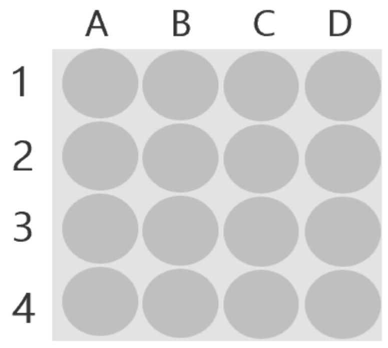


*Figure 32 Simplified area with four mini-columns with four cells each*

Figure 33 shows how the TM learns the sequence of elements. The figure shows two sequences: ABCD (green) and BDAC (blue), with their so-called *Temporal Path*. The *Temporal Path* shows temporal dependencies between SDRs representing particular sequence elements. Every learning cycle, $t_0$, $t_1$, $t_2$ and $t_3$, represents a cortical model (grey box) defined in Figure 32. In every cycle, an element of the sequence is presented to SP. The SP encodes the element into a set of active mini-columns. This simplified example encodes the element with a single active mini-column. Element 'A' is encoded by activating the mini-column A, element 'B' by activating the mini-column B etc. For example, in cycle $t_1$, element B of the first sequence (green) is spatially encoded by the second mini-column B. In the same cycle, element D from the second sequence (blue) is encoded by the fourth active mini-column D.

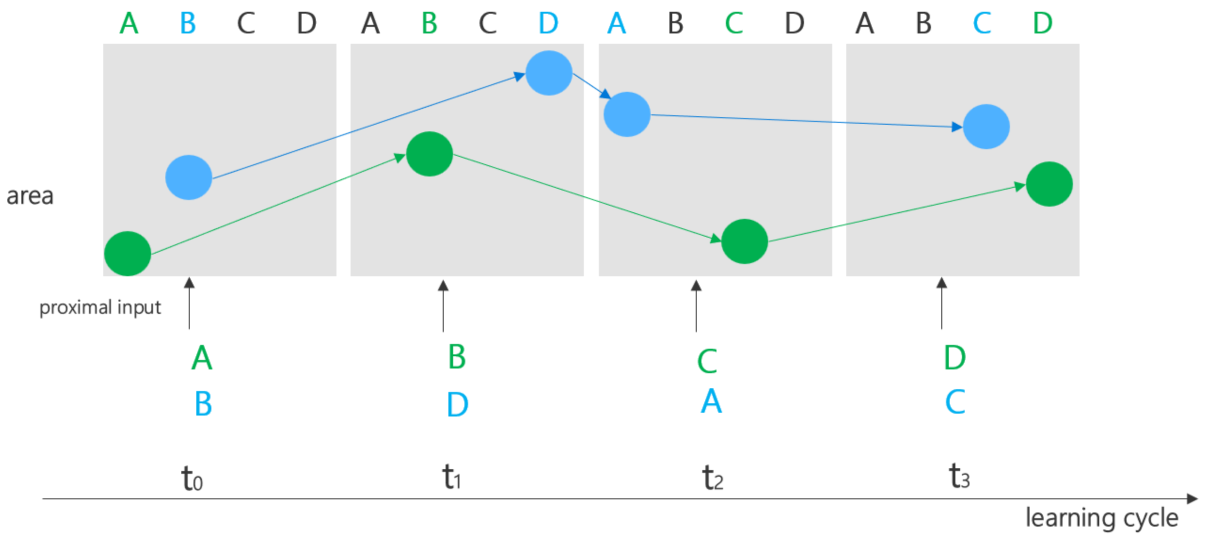


*Figure 33*

*Temporal Learning in simplified form. An active cell represents the set of active cells. Active mini-column represents a set of active mini-columns.*

At the beginning of learning in cycle $t_0$, the algorithm chooses the set of active cells by Algorithm 11 (Lines 19-27). When learning the first sequence, this set is represented as a green cell in the active mini-column A. Analog, in the case of the second sequence, this set is represented as a blue cell in the active mini-column B. In the real, non-simplified model, every active cell in Figure 33 would represent a set of active cells, and every active mini-column would represent a set of active mini-columns.

Further activation of cells in the learning process is driven by Algorithm 12. After a few iterations associating synapses are created between cell populations that define a set of active cells for states in $t_0$, $t_1$, $t_2$ and $t_3$. The temporal synaptic path is established after the learning process is completed, as shown in Figure 33. The synaptic associations depolarize associated cells as predictive cells. Predictive cells are cells that are supposed to be active in the next iteration. Having the population of predictive (depolarized) cells, it is also possible to look up the set of predictive cells in any other upcoming iteration. With this mechanism, the TM can predict the population of active cells in any iteration in the future.

# 6 The concept of Distributed Compute of mini-columns

As described in Chapter 3.1, both NAA and HTM CLA are dedicated to reverse engineering and replicating the neocortex's functioning. One of the trending issues in this research was, from the beginning, the support for the large-scale model with a large number of cells and synapses. If the cortical algorithm requires many cortical areas to execute a needed calculus and increase cognitive capabilities, the problem of distributed computation must be solved. This chapter describes the distributed model of the NAA algorithm with a single area corresponding to the HTM CLA.

## 6.1 Introduction

The primary implementation of HTM CLA was initially implemented in Python as a part of the NUPIC framework developed by Numenta. (Numenta, 2008). Also, C++ and JAVA implementations of HTM CLA are available (open-source, htm-java, 2013). Because many modern enterprise applications use .NET with increasing demand for cross-platform support (Linux, Windows and macOS) for professional enterprise developers (Anon., 2019), the NAA and an extended version of HTM CLA (Dobric, 2019) were implemented to support this research. By design, the .NET version of HTM CLA mainly aligns with JAVA implementation, initially motivated by the Python version. This design decision makes it possible to compare experimental results across different implementations. All algorithm changes, and additional features in the new framework are implemented as an extension.

The new framework initially supported only a single-core computation for Spatial Pooler and Temporal Memory algorithms (see Chapter 5). It didn’t support multiple CPUs and computation at multiple nodes. This chapter describes how the Spatial Pooler was successfully redesigned for scale by supporting multiple processors at a single machine and the parallel execution in a distributed environment.

This work started with the realistic assumption that the algorithm described in this work configured with a high number of neurons is CPU and RAM-intensive. As a result, all algorithms described in Chapter 5 are relatively to standard algorithms (i.e., back-propagation) straightforward and efficient. However, the complex topology of areas in

HTM requires more computing power and memory in some scenarios than on a single "*commodity*" machine.

To support the required scale, the original version of the algorithm had to be redesigned. The investigation has shown that the original design has, in this context, two limitations of interest: The memory limit of the synaptic matrix and the long calculation time required due to a large number of synapses.

For example, in the case of 4096 mini-columns connected to 1024 input neurons, which corresponds to an input image of 32x32 pixels, the SP algorithm will create 4.194.304 synapses using the global inhibition described in 5.7.3.1. The challenge of this task was not only to redesign the existing algorithm for distributed calculation. The goal was to maintain the existing concept and ensure alignment with corresponding implementations in other frameworks.

The following sections describe how the cortical algorithm on the example of the Spatial Pooler (5.7) was redesigned to run on multiple CPU cores and nodes and run on a theoretically unlimited number of mini-columns and areas. The redesign's emphasis was on using the Actor Programming Model described in section 4.2. After investigating several parallel techniques described in Chapter 4, the Actor Programming Model (4.2) was chosen as the best approach. The new, redesigned Spatial Pooler supports the elastic distribution of computation inside a single area and connects multiple regions distributed in a collective cognitive network.

## 6.2 Making the proper Actor Model Framework

Both approaches, HTM CLA and the Actor Programming Model (APM), are biologically inspired models. This abstraction naturally integrates both models. The promise of a new design on top of the APM reasoning is the simplification of the distribution of computations across different cores, especially across different machines. The HTM CLA is a complex algorithm with many interconnected synapses, dendrites and cells. This work aims to simplify the design of distributed cortical algorithms by using the Actor Programming Model (APM). Because the existing framework *neocortexapi* needs to be extended, an Actor Model implementation is required, which shares the same underlying platform .NET.

All frameworks described in section 4.2 are well-designed and successfully applied in the industry. Still, they mostly do not offer custom partitioning functionality or rely on some corporate-specific product (i.e., Microsoft Service Fabric). For example, the Orleans framework was the original Microsoft Actor Programming Model framework. After a while, the company decided to implement the new one called the Service Fabric. Both frameworks simplify the implementation of distributed systems and make all intricate details transparent to developers. However, needed changes in the underlying algorithm are difficult or not possible.
Moreover, due to frequent changes in the strategy of approaching the best solution, the most promising framework appeared Akka.NET. As described in 4.2, all the listed frameworks have shown unsatisfactory results, such as high CPU usage, a lack of control over state persistence and craching due to high memory consumtion.

Unfortunately, after performing the first experiments, it has also manifested unsatisfactory performance results regarding networking under a high CPU load and custom persistence of the state. Also, using the Serverless Actor Model inside Durable Entities technology was not a good option for scientific work. The approach, *simplicity over flexible extensibility,* does not allow required changes or controlled clustering in the framework. Therefore, the dependency on the serverless model, in general, is insufficient for this task.

Finally, all listed frameworks are designed to build enterprise state-full services efficiently. They are designed to be simple and efficient for widely used enterprise scenarios. However, one of the critical requirements of this task was to move a part of the routing and partitioning functionalities (see 4.2) to the code that sends messages to the actor. To share the load efficiently across all nodes in the cluster, the Actor Model needs to know about the topology of the cortical algorithm. This detail is only known to the algorithm. This requires an Actor Model framework with the ability to make routing decisions outside of the actor server, violating the Actor Model reasoning principles. With the ability to place the computation logic on the right node within the algorithm, higher efficiency of overall CPU power across multiple nodes can be achieved. For this reason, a lightweight dedicated Actor Model Framework was designed and implemented in C# .NET Core (Chadha, Dobric, 2020).
Actor Model Frameworks commonly use location transparency (see 4.2) when executing code on remote nodes. This is a valuable principle because it simplifies

implementing the APM code. Unfortunately, complex algorithms like cortical algorithms cannot rely on this concept for efficiency reasons. The locally executed part of the algorithm has the best knowledge of the current execution state. Therefore, it can make the best decision on where to execute which part of the algorithm. This concept enables the local part of the algorithm to place the execution of the code anywhere in the cluster. It introduces a dedicated partitioning concept that extends the classical Actor Programming Model. It uses a logical partition, which defines an isolated unit for executing complementary computational steps. With this, the algorithm does not decide about a thread or physical node, which would require more complex code. Instead, it uses a logical concept of partitions organized as virtual containers inside the host process. It can choose a partition as a logical artefact but remains location transparent. Before the algorithm runs, the initialization process creates a logical representation of partitions and associates them with the physical node.

The framework developed to support this research (Chadha, Dobric, 2020) is based on the existing messaging platform provided by Microsoft Azure Service Bus (Anon., 2009). Please note that the messaging framework can be replaced by any other one that supports the client message session concept (Pelluru, Schneider, Wolf, 2021). The session concept ensures that a single message receiver will receive messages with the same session identifier at a single place in a cluster. This means that it is guaranteed that messages with the same identifier will not be received by more than one receiver anywhere in the cluster. This lightweight framework combines the Remote Procedure Calls (RPC) and the Message API styles into a single framework. It uses the RPC API style to enable easy and intuitive procedural implementation. At the same time, it keeps the internal messaging transparent to the developer to simplify the usage of the framework. The Message APIs style uses a broker to distribute messages to nodes in the cluster and enable more effortless adding of new nodes while the system runs. The following pseudo-code shows how to start the execution of the calculation of the overlap of a column within the algorithm.

First, the actor's local system is instantiated (Line 03), which drives the calculation and will play the role of scatter described in section 4.2. Next, multiple system instances can be instantiated, which acts independently. Then the proxy to an actor is created (Line 05). Finally, the Actor itself is executed remotely on some of the nodes.

The executing code in this context implements some calculus of the Cortical Algorithm. Finally, the calculus is started using a generalized method Ask (Line 07). This method routes the message request to an actor on its node in the cluster, starts the calculation and awaits the result. Combining the actor type name and the actor identifier is used to create the unique message session identifier. With this rule, every actor receives messages in a dedicated message session, ensuring that the actor code runs at a single node. The class *HtmActor* implements many different calculations as remote operations (*Overlap*, *AdaptSynapses*, *BoostColumns*, etc.), which are looked up by messages named *Overlap*, *AdaptSynapses*, etc., respectively.

Listing 1  Example of pseudocode that invokes an actor compute

```
01| function runactor_sample (name, config)
02|     // create the proxy to the actor system
03|    $s$ ← ActorSystem(name,config)
04|     // create the proxy to the actory system
05|    $a$ ← $s$.CreateActor (HtmActor, colIndex)
06|    // Invokes the actor remotely with argument “overlap”
07|    res ←a.Ask(“Overlap”)
08| end
```

However, before an operation in the actor can be invoked, the Actor Model Cluster needs to be initialized. To create the cluster (host) running theoretically any number of actors, the pseudo-code in Listing 2 is used. The function *runNodeInCluster* is supposed to be started inside of a process. It implements the Actor Host algorithm that theoretically runs on any number of nodes. The system scales by increasing the number of hosting processes that form the cluster of actors. Every Actor Host automatically loads and maintains requests dedicated to the actors. The proposed model builds an infrastructure that can run in any container, making it suitable for execution in modern cloud environments. The cluster node starts at line 3 simultaneously at multiple physical machines. Nodes in the cluster do not have any knowledge about each other. Every node is connected to a dedicated subscription defined by its name *subscriptionName* under the given topic.

The topic in this context corresponds to the standard queue listener implemented by the broker. Hence, a topic implements a message subscription pattern. It is an extended implementation of the *Queue* – Point to Point pattern (Sachs Kai and Appel, Stefan and Kounev, Samuel and Buchmann, Alejandro, 2010) called Publish-Subscribe (Pub/Sub) (Mishra Tania Banerjee, Sahni Sarta, 2013).

Listing 2 Pseudo-code executed at all hosts.

```
01| function runNodeInCluster
02|   $s \leftarrow$ ActorSystem(name,config)
03|   $s$.Start (config.SubScriptionName)
04| end function
05|
06| function Start(subscriptionName)
07|   DO
08|       $s \leftarrow$ AcceptMessageSession(subscriptionName)
09|       runMessageReceiverAsync(s, isStopRequested)
10|   WHILE isStopRequested = FALSE
11| end function
12|
13| function runMessageReceiverAsync(session, isStopRequested)
14|   DO
15|     $m \leftarrow$ session.ReceiveMessage()
16|     $result \leftarrow invokeActor(m)$
17|     $sendResult(result)$
18|   WHILE (isStopRequested)
19| end function
```

Every *Topic* hosts physically multiple receiving *subscription queues* called *Subscriptions*. That means that every message sent to a *Topic* is copied to the *subscription queue* defined under the topic as long the *subscription* filter matches. Each *subscription* in this framework defines a *filter expression* corresponding to the node's name, which hosts the Actor System at the node. When the actor proxy (Listing 1, Line 07) converts the procedure call to the message, it attaches the name of the destination node to the message, so the filter at the appropriate node can peek at the message. For example, the code shown in Listing 1 knows about nodes running in the cluster. This implementation enables the code on the client side to selectively choose the message's route and implicitly place the computation on the remote node inside the selected actor. The algorithm in the scatter can decide at which node the code will be executed by setting the subscription name. Selecting the dedicated subscription violates the location transparency principle (see 4.2) of the Actor Model, but it helps to increase the performance of the cortical algorithm.

Accepting a massage session in line 8 is coordinated by the underlying messaging system of the Service Bus broker. It automatically accepts a set of messages from the same session. Once the message session is accepted, it remains in the same process during the entire calculation time of the actor. For example, if some calculation is driven by N messages (several steps), all calculation parts triggered by N messages remain in the same process. With this in mind, the Actor Model will ensure that all messages inside the same session S1 will execute sequentially on the same node and in the same process. In contrast, for all messages which belong to another session, S2 will execute parallel to S1.

This concept provides flexibility to run any code remotely and can easily decide which code (actor) can execute sequentially or in parallel. Independent of this research, the introduced Actor Programming Model framework can be used for the implementation of any parallel algorithm.

The following section describes how this framework helps to model the distributed CLA inspired by HTM and NAA.

## 6.3 Distributing a Partial Computation

An Actor Programming Model framework described in the previous section enables the straightforward distribution of the computation by taking complete control of the computation that will be executed in a sequence or parallel.
Assume some complex computation can formally be generalized as a set of sequential partial computations expressed by the following equation,

$$\mathrm{C} = \{c_1, c_2, .., c_Q\} \tag{41}$$

where $\boldsymbol{Q}$ defines the number of partial computations. The computation set $\mathrm{C}$ is implemented inside the actor code, let's say actor A, as a set of operations that will be executed remotely. Algorithm 15 starts at Line 05, using the actor *local system* defined in Listing 1 to initiate the orchestration function $\varphi$. During the initialization process, the local system will connect to the cluster of $N$ nodes and associate every actor instance to a dedicated message session $P$ running each at the dedicated node. Every partial computation in the set $\mathrm{C}$ is typically executed sequentially using the orchestration function $\varphi$. The function $\varphi$ spans a distributed computation $c_i$ that is executed in multiple instances of the actor A in the cluster (Lines 06-09). The parallel execution is started by function φ (Line 07) using a partitioning algorithm described in the next section. The function φ will invoke the remote parallel computation $c_i$ on multiple instances of actor A each at a dedicated physical node. Every actor instance is associated with the dedicated message session. This approach guarantees that every instance of the actor runs only once in the cluster, as described in a previous section. Note that the function φ does not implement the computation logic. It acts as a proxy and routes the computation request to the concrete implementation φ′, which runs remotely.

Algorithm 15 Partial Computation

```
01| input: I // Set of neural cells. I.e. sensory input.
02| output: o // Set of neural cells. I.e. active cells of a mini-column.
03| configuration: N //Nodes, P // Sessions
04| begin
05|   $\varphi \leftarrow \text{actorSystem}(\complement, N, P)$ // Creates an orchestration function
06|   FOREACH $c_i \in \complement$
07|       $r \leftarrow \varphi(c_i, I)$           // Runs in parallel for each $p \in P$
08|       $o_i \leftarrow S(o_{i-1}, r)$        // Recalculate internal column state
09|   ENDFOR
10|   return o
11| end
```

Finally, the function $S$ in line 8, which is usually not CPU intensive, recalculates the internal state depending on the results of partial calculations in Line 07.

The cluster of nodes that run the Actor Model is created by running a wanted number of identical processes, which executes the code shown in Algorithm 16. Nodes in the cluster are not interconnected and run independently of each other. This is not required because the execution logic is controlled at the node that initiates the distributed execution. The actor systems start at every node with the initialization of message receivers for the limited number T of concurrent sessions P (Lines 01 and 02). Line 04 initiates the listening for incoming computation requests. Messages received by the listener contain the requested computation $c_i$, the partial computation identifier $\boldsymbol{i}$ and the actor identifier $id$ including the set of arguments $args[]$ dedicated for the $c_i$.

The identifier $id$ in Algorithm 15 is the index of the mini-column, which suggests the partial implementations of a mini-column as an actor. Because the function φ (Line 5, in Algorithm 15) holds the session P, all messages sent to the actor $a$ with the same identifier $id$ will be processed by $\varphi'$ in order. A next (line 6) the $thread^t$ is created.

Algorithm 16 Remote Execution of the partial computation in the cluster

```
01| input: actorSystem        // Actor system initialization.
02| configuration: P, Ç, T  // P:sessions,  Ç: Cache, T: max concurrent actors
03| begin
04|         $(thread^t, c_i, id, \varphi', args[]) \leftarrow actorSystem(msg) \mid c_i \in \c{c}, 0 < t < T$
05|         begin thread
06|         $thread^t(response, c_i, id, args[])$
07|            IF $c_i \notin Ç$ // If computing logic is not in the cache
08|               $Ç \leftarrow Ç \cup (a \leftarrow [\varphi'(c_i, id)])$  // Create the actor and add it to the cache
09|            $a \leftarrow Ç(id)$ // Get the actor with id from the cache
10|            $response \leftarrow response \leftarrow a(args[])$    // Execute the compute logic
11|         end thread
12| end
```

According to Actor Programming Model rules, there is a single execution of the partial calculation of $c_i$ running in a unique thread for every actor instance in the entire cluster. The mini-column actor with the partial computation $c_i$ is created by the function $\varphi'(c_i)$ (line 08) or retrieved from the cache (line 09), if previously already created. The same orchestration function $\varphi'(c_i)$ prepares the computation result in Algorithm 15 - Line 07, which has requested the partial computation.

This approach makes it easy to distribute any computation logic that can be partitioned.

## 6.4 Redesign of the Spatial Pooler

The SP and TP default design was initially migrated from JAVA to NET Core, supporting single-threaded execution of both algorithms. This alignment enables easy debugging and investigations of the cortical algorithm under various conditions. However, due to single-core limitation, it is constrained when applying the algorithm in real-world applications. Therefore, the SP algorithm was redesigned in this work to support parallel execution.

In the first step, it was investigated how the SP can support multiple CPU cores when running inside a single process on a single machine. It was clear that running parts of an algorithm on multiple cores on a single machine is still a different and more challenging approach than running the same on multiple physical machines. However, the first step in both approaches was the same. The task was to identify possible partial calculations of the algorithm, which can be contextually split from each other. This section describes identified structures of the original design of HTM and SP, intending to redesign them and enable them for parallel execution.

The matrix $\lambda$ defined by equation (24) enforces the design of the SP to be tightly coupled to a single thread and a single process. This matrix represents a *shared resource* that requires controlled concurrent access (4.1). Threads are necessary to speed up the execution, but they require concurrent access to the shared matrix, which is a singleton instance in this algorithm. That means that even if some thread does not need to access all matrix elements, the whole matrix must be locked by a single thread for a short CPU time, usually called quantum time. For example, if all mini-columns are executed in parallel, every mini-column will need to access only a single row of the matrix (see 5.7). Every computation $c_i$ inside a mini-column is a part of the SP algorithm that can be executed independently in the context of the mini-column. In other words, the idea is to refactor the code into blocks that can run independently. The calculus inside of the mini-column $C_1$ can be executed independently on the calculus of the mini-column $C_2$ (see Algorithm 2 and Figure 24). In that case, mini-column computation can run independently.
This matrix was removed from the original version of the SP and semantically restructured into the graph inside mini-columns to better share the memory and enable partitioned calculus of the entire cortical space. Figure 34 shows a mini-column that integrates the part of the model in Figure 23 related to the SP.

Removal of this matrix enabled more straightforward reasoning about a single mini-column calculus, which fits better with the proposed Actor model and biology of the mini-column. With this redesign, it is possible to separate the SP algorithm into independent mini-columns and distribute them across multiple nodes. Furthermore, if the algorithm that runs inside mini-columns is implemented as an Actor, then using distributed locks can be omitted. After the suggested redesign, three implementations of SP are implemented and considered:

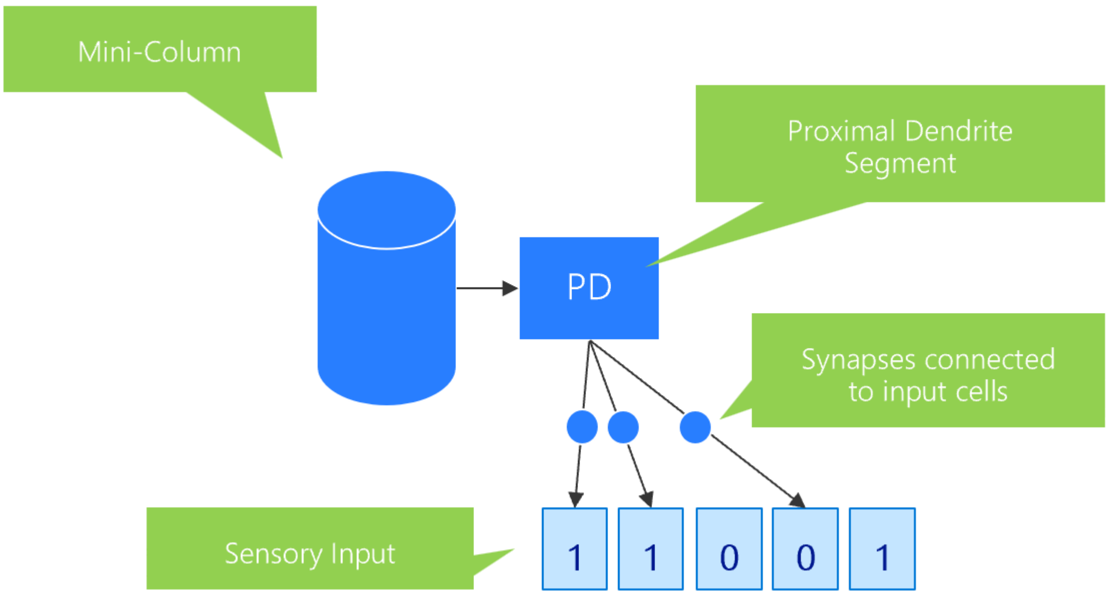


*Figure 34*
*Simplified representation of a single mini-column in the NAA and HTM model related to the Spatial Pooler.*

Single-Threaded Spatial Pooler

The single-threaded SP is the original version of the SP without algorithm-specific changes. This version aligns with the initial referenced implementation of HTM CLA as described in 3.1. This version of the SP is suitable for research and debugging. However, it is limited to execution at the single CPU core only. Because of this, it shows a lack of performance and is incapable of parallel execution.

Spatial Pooler Multithreaded

This version of SP was designed and implemented as a first redesigning step of a referenced single-threaded version. It supports multiple CPU cores on a single machine. This version does not use the Actor Programming model and cannot be used to run in parallel at various nodes in the cluster. However, the redesigned HTM model and the Spatial Pooler in this version are conceptual models for the parallel version SP-Parallel.

Spatial Pooler-Parallel

SP-Parallel is the parallel version of SP designed and implemented in this work. It uses the SP-MT version's conceptual model and supports multiple cores and nodes on top of the Actor Programming Model (4.2). The rest of this chapter describes the core concept of this version.

In the context of parallel implementation, the Spatial Pooler algorithm consists in general of two critical stages that must be considered separately. Initialization and Learning stages are grouped in partial computations (partial algorithms) shown in Figure 35. For example, the *Initialization* stage prepares the parallel computation. Mini-Columns and synapses in the initialization stage create mini-columns cells, proximal dendrite segments and synapses between segments and input cells. The *initialization* stage runs once, and the *Learning* stage runs for every spatial input.

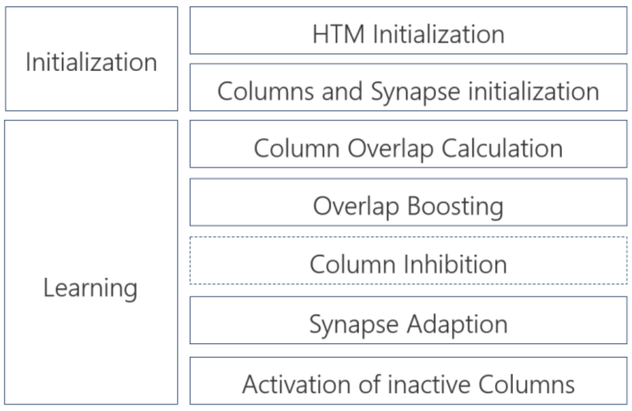

Initialization
HTM Initialization
Columns and Synapse initialization
Learning
Column Overlap Calculation
Overlap Boosting
Column Inhibition
Synapse Adaption
Activation of inactive Columns

*Figure 35*
*Spatial Pooler logic is grouped into Initialization and Learning stage (on the left). Every stage has corresponding partial calculations (on the right).*

Except for *Column Inhibition* (Figure 35 right) introduced in 5.7.3, which is currently shared across all three versions of the SP, all other computations are SP-version specific. In future versions of the Spatial Pooler, the inhibition of mini-column and possibly other parts of the algorithm (not listed in Figure 35) might also be redesigned for parallel execution.
They have importance from the engineering point of view, but their redesign would not scientifically contribute to this work.

The Spatial Pooler Parallel focuses on partitioning memory and CPU usage. The memory consumption inside SP can be expressed as follows:

$$m = m(i_k) + \sum_{u=1}^{M} m_c(u) \tag{42}$$

$$m_c(u) = m_{0c} + \sum_{w=1}^{Q} m_s(w) \tag{43}$$

Where:

M – Number of mini-columns.
$m$ – Memory consumption of a single SP instance during learning.
$i_k$ – The spatial input as produced by encoder: $k\epsilon\{1, N\}$
$m(i_k)$ – Memory consumption of an input vector.
$m_c(u)$ – Memory consumption of a mini-column.
$m_{0c}$ – Memory of a mini-column without synapses. Nearly the same for all mini-columns. It depends on references to a different count of synapses.
$m_s(w)$ – Memory occupied by an instance of a synapse inside a mini-column. The part of the equation (43) corresponds to the memory occupied by the proximal dendrite segment with $Q$ synapses.

The reference implementation of the SP and the Multi-Threaded version consumes the same overall memory $\boldsymbol{m}$ inside a single process. For this reason, they are limited by the physical memory of the hosting machine. Mini-column instances and their synapses occupy a majority of the memory. Every row of the matrix $\boldsymbol{\lambda}$ is by new design associated with the mini-column (see Figure 23). By sharing mini-columns and associated computations across multiple physical nodes, the overall occupied memory at the single node will be reduced according to equations (42) and (43). The challenge here is to find the right way to group mini-columns and place them on physical nodes, including the code required to operate on columns.

The SP-Parallel version solves this problem using a dedicated partitioning technique to allocate the required space and execute code remotely.

## 6.5 The Concept of Partitions

As mentioned in the previous section, the execution can be processed in parallel on multiple CPU cores (SP-MT) and parallel on multiple nodes (SP-Parallel). To ensure efficient parallel execution, mini-columns must be shared between nodes in the cluster where the algorithm will be executed. Because the number of mini-columns is typically much higher than the number of physical nodes, an algorithm must be created which calculates the proper distribution of mini-columns across nodes. The process of distributing mini-columns is called partitioning, and the algorithm that performs this task is called the partitioning algorithm in this work. All mini-columns that belong to different partitions will be processed in parallel across multiple nodes. For example, ten mini-columns will run in parallel on multiple nodes if associated with a different partition.

In contrast, mini-columns within the same partition will be processed sequentially. Theoretically, the maximal performance can be achieved if all mini-columns run parallel. Running all mini-columns in parallel is possible if the total number of CPU cores across all nodes in the cluster equals the number of mini-columns M and if every single mini-column is associated with a single partition.

The SP-Parallel creates partitions only in the initialization phase. The partitioning algorithm runs on the chosen scatter node (see 6.26.2) that controls partitioning, invokes remote partial calculations and collects results. The partition is defined as a set of tuples which describe how to execute remote code. A partition tuple holds the following elements: the Partition Identifier, a set of indexes of mini-columns assigned to the partition, and the actor's descriptor, which will host mini-columns.

The initialization stage of the SP calculates and creates partitions and then starts placing partitions on physical nodes. The current implementation of SP-Parallel uses the uniform partitioning algorithm shown in Listing 3. The purpose of the function *createPartitions* is to distribute mini-columns across available physical nodes in the cluster uniformly. Assuming that `numElements` is the total number of available mini-columns in the area, the algorithm first calculates how many columns can be placed in the single partition (Line 04). The algorithm ensures that all partitions are uniformly distributed across all nodes in the cluster.

For example, given the total count of mini-columns `numElements` = 90000, the number of nodes = 3, and the number of partitions `numOfPartitions` = 35, 34 partitions will be filled with 2572 elements. The last partition will be filled up with the remaining 2552 elements. This algorithm assumes that all mini-columns associated with partitions will have similar execution times. This holds because the mini-column boosting algorithm section 5.7.4 inside SP ensures that all columns are uniformly engaged in the learning process. The slight difference in the mini-column calculation time depends on implementing homeostatic plasticity, as discussed in section 5.7.4.

Listing 3 Pseudo code that demonstrates uniform creation of partitions

```
01| createPartitions(numElements, numOfPartitions, nodes[])
02|
03|     $destNodeIndex \leftarrow 0,\ min \leftarrow 0,\ max \leftarrow 0$
04|     $numPerPart \leftarrow \left\lfloor \frac{1+numElements}{numOfPartitions} \right\rfloor$ ; $numElements > numOfPartitions > 0$
05|
06|    FOR $partIndx = 0$ TO ($numOfPartitions$ OR $min$ >= $numElements$)
07|       $min \leftarrow numPerPart * partIndx$;
08|       $maxPartEl \leftarrow numPerPart * (partIndx + 1) - 1$
09|       IF $maxPartEl < numElements$
10|          $max \leftarrow maxPartEl$
11|       ELSE
12|          $max \leftarrow numElements - 1$ // filling the last partition
13|       ENDIF
14|       $destNodeIndx \leftarrow destNodeIndx \,\%\, |nodes|$
15|       $placement \leftarrow (destNodeIndx, partIndx, min, max)$
16|       $map.add(placement)$
17|       $destNodeIndx \leftarrow destNodeIndx + 1$
18|    ENDFOR
19|    return map;
20| end
```

Next, the algorithm traverses all mini-columns (Line 06) sequentially. It uses the flattened index of the mini-column according to equation (22). Then, it looks up the mini-column range flatten index (min-max index Lines 07-12) and associates it with the partition and the node (Lines 14-16). The result (Line 19) is a so-called *placement* set that maps mini-columns to partitions associated with nodes. The algorithm in Listing 3 is not HTM-dependent. It can be used for any other kind of distributed calculation.

Another important task was investigating how the calculation described in Chapter 7 can be remotely executed in the context of a cortical algorithm. This work focused on the SP algorithm described in section 5.7. Using the same approach, the Neural Association Algorithm (see 5.8) can also be adapted for parallel execution. The adoption of the SP corresponds to the parallel implementation of Algorithm 5. Every partial calculation in Figure 35 performs calculations over columns and synapses. They all were redesigned to operate on a single mini-column instead operating over the entire matrix $\boldsymbol{\lambda}$.

A partition placement algorithm is created according to the described partitioning mechanism in Listing 3. In the case of node parallelization, the computation of every mini-column can theoretically be executed on any core. However, the partitioning algorithm places a set of mini-columns in a single partition. In simplified form, the overall calculation time can be defined as follows:

$$t = M * t_s + \frac{1}{N_c} \sum_u^M t_u + M\, t_g \mid m < m_\theta \tag{44}$$

Theoretically, the total time for any step (see *Figure 35*) is defined as the sum of scatter time $t_s$ needed to dispatch the computation remotely, the sum of all mini-column computations $t_u$ divided by the number of cores $N_c$ and gather time $t_g$ for collecting results. Note that the mini-column time $t_u$ depends on the number of connected synapses on a proximal segment. This equation assumes that the total working memory of the node is below the threshold $m_\theta$. If this threshold is exceeded, the operative system will reallocate memory fragments to the disk. This effect is called *hard-page* fault and typically leads dramatically drop in performance. Developers should take care of proper configuration to avoid this effect. Computation time in such a distributed system is more complex than shown in the previous equation:

$$t = t_{rcv} + t_{sched} + t_{start} + t_{calc} + t_{persist} + t_{end} + t_{send} \mid m < m_\theta \tag{45}$$

*rcv:* Time for receiving the message, which triggers the calculation

$t_{sched}$: Time required by the system to schedule calculation.

$t_{start}$, $t_{end}$: Time overhead to start and end the computation inside the framework.

$t_{persist}$: Time to save the current state of calculation.

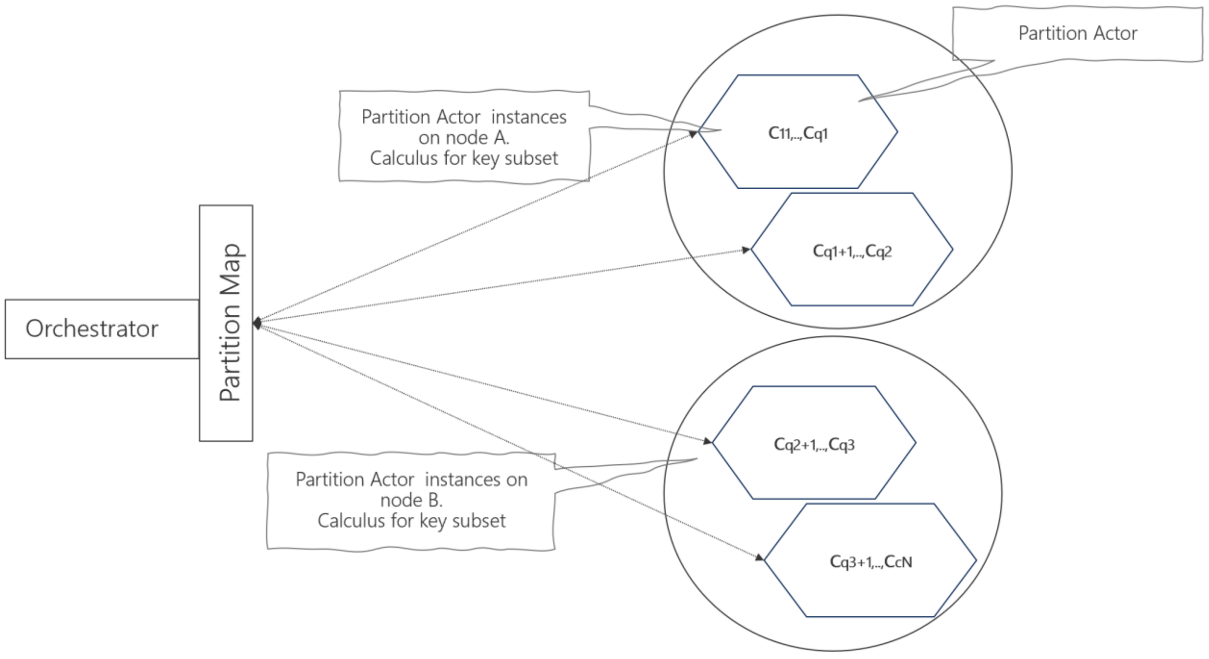
*$t_{sched}$: Time to send the result.*

Coordinating the lifecycle of partitioned calculations is not a trivial task. Messages must be ordered, and distributed locks avoided whenever possible. To solve this problem, the discussed Actor Programming Framework was used (Chadha, Dobric, 2020). The framework extensively uses the so-called message session concept. The message session concept ensures that all messages inside the same session are executed in the order they arrive. The named framework associates the partition with the session. In that case, every message sent inside the same session is processed inside the same partition. With this design decision, partial computations defined by (41) are guaranteed to be executed in order without overlap within the same node. Computations are orchestrated by Algorithm 15 and performed by Algorithm 16. This design ensures that every single computation runs in a single thread in the entire cluster. Because of this, no distributed lock is required. The time $t_{sched}$ is, in this case, minimal. Therefore, it is removed from the algorithm and remains part of the messaging system.
In this concept, one partition is implemented in the Actor. The Actor defined by the partition is responsible for executing the computation over multiple mini-columns in the sequence. The mini-column computations that belong to other partitions (actors) will run in parallel. Every mini-column performs computation over RF space $X_k$ as defined by equation (23). This space is much smaller than the entire space $\boldsymbol{\lambda}$ and enables the faster execution of the algorithm.

$$X_k = \{x_{k1}, x_{k2}, .., x_{kU}\} \,|\, U < N * M \tag{46}$$

The partition Actor operates on a set of $X_k$ elements defined by placements calculated in Listing 3, and it runs every partial computation listed in Figure 35. The distributed algorithm of SP-Parallel performs partitioning inside the orchestrator node (see Figure 36 left), which plays the role of the *scatter* where the partitioning algorithm (Listing 3) is executed. Actors on remote nodes are started (orchestrated) by routing messages to the messaging broker (see Listing 1) initiated by the *Ask* function (Line 07) and awaited on multiple threads inside of the orchestrator.

The Actor model system runs session listeners (see 6.2) that receive messages (Listing 2, Line 08) from the dedicated topic subscription associated with the physical node. Every accepted session executes actors, and results are sent back to the orchestrator node.



Partition Actor
Partition Actor instances on node A. Calculus for key subset
C11,..,Cq1
Cq1+1,..,Cq2
Orchestrator
Partition Map
Cq2+1,..,Cq3
Partition Actor instances on node B. Calculus for key subset
Cq3+1,..,CcN

*Figure 36*

*Partitioned column space. Every partition is implemented as an Actor, which owns a mini-column subset. Given the number of mini-columns and nodes, the number of partitions can be chosen.*

To recap, the orchestrator knows the number of partitions and nodes in this partitioning concept. As discussed, this is the needed change in the location transparency definition of the Actor Programming Model. The SP-Parallel orchestrator code places a partition to the required node. With this concept, the Actor Model framework ensures that the computation is executed in a sticky session on the node. A sticky session improves performance and does not require a durable persistence of the calculation state because the state is kept in the cache.

Another approach was also tested but excluded from this work, where the orchestrator node operates without any knowledge of the cluster.
Such design enables a simpler system architecture but requires a durable state. This means that nodes must be able to durably load and save their state, which adds significant performance costs defined by $t_{persist}$. The second approach performs better in typical enterprise applications with shorter computation time and smaller states. In contrast to typical enterprise applications, HTM CLA requires longer computation and large memory consumption for state persistence due to a large number of cortical columns and synapses.

## 6.6 Results

In this work, various experiments have been conducted to compare the performance of redesigned Spatial Pooler algorithms, namely SP-Parallel, SP-MT, and the reference implementation. MNIST (Yann LeCun, n.d.) images of 28x28 pixels have been used for all tests. Initially, the performance of the single-threaded algorithm was compared to the SP-MT using various topologies (results presented for 32x32 columns). Subsequently, the computation time of SP-Parallel was evaluated for a column topology of 200x200 on one, two, and three physical nodes.
Most experiments today use HTM column topologies with 2048 or 4096 mini-columns, which can be executed on a single node. Finally, the performance of several topologies has been tested in a cluster of the three physical nodes.

All tests have been executed on nodes with the following "*commodity*" configuration on virtual machines in the Microsoft Azure cloud:

| | |
|---|---|
| Used OS | Windows Server 2016 |
| CPU | Intel Xeon E5-2673 v4, 2.30GHz, 2295 MHz, 2 Cores, 4 Logical Processors |
| RAM | 16.0 GB |

The initial experiment aimed to demonstrate the performance enhancements of SP-MT compared to the single-threaded SP algorithm, both operating on a single node. The experiment utilized the MNIST test image of 28x28 pixels and employed a cortical column topology of 32x32. Figure 37 shows the resulting sparse representation of the MNIST image.

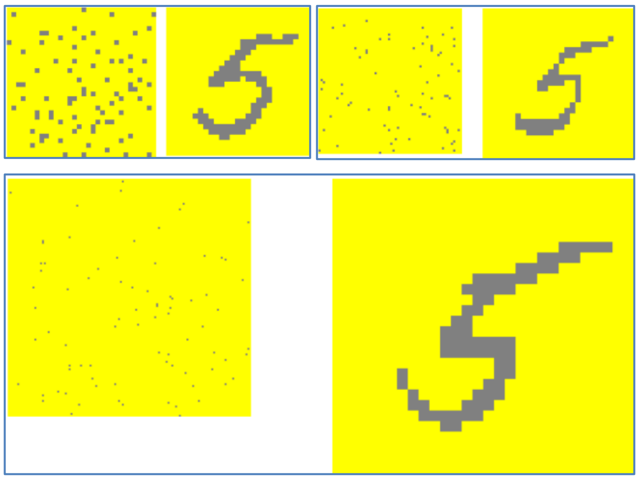


*Figure 37.*
*NIST digit in sparse representation. Column topologies are 32x32 (top-left), 64x64 (top-right) and 128x128 (bottom). Every topology can result in different sparsity. For example, the image with a topology of 64x64 has a sparsity of 2%, which corresponds to  (81) mini-columns.*

Based on Figure 38, it can be observed that SP-MT achieves a speed approximately two times faster than the single-threaded SP on the specified configuration.

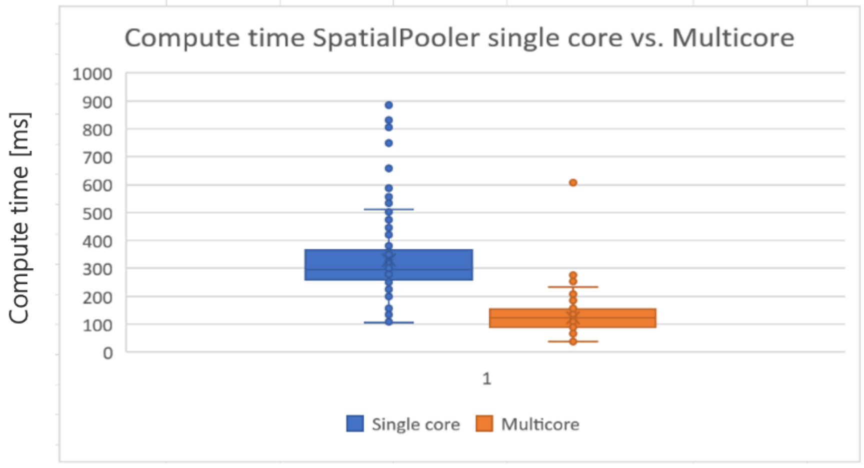


*Figure 38.*
*Performance results, SpatialPooler single-core (SP) versus Spatial Pooler multicore (SP-MT) on a single machine with MNIST 28x28 test image used 32x32 columns.*

During the same experiment, measurements were taken to assess memory consumption (see Figure 39) and processing time in relation to the column topology (as depicted in Figure 40).

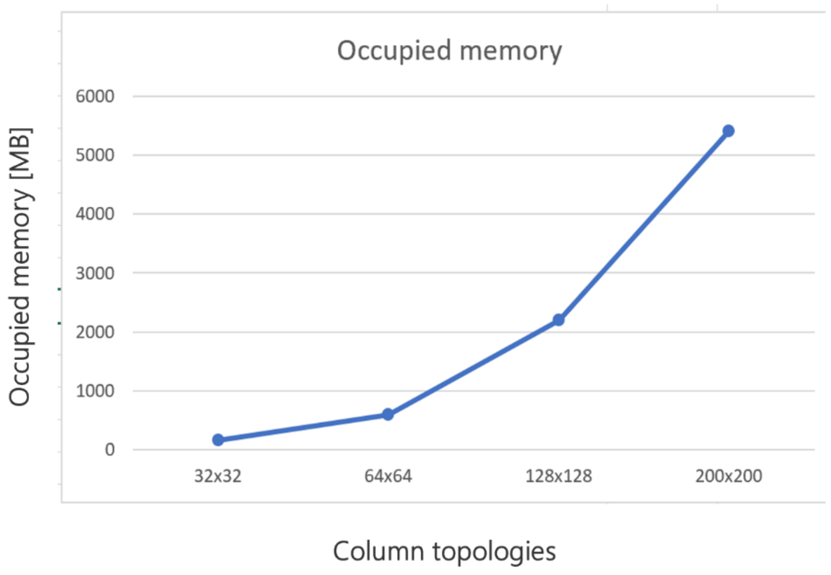


*Figure 39*
*Memory usage of the SpatialPooler-MT algorithm on a single node.*

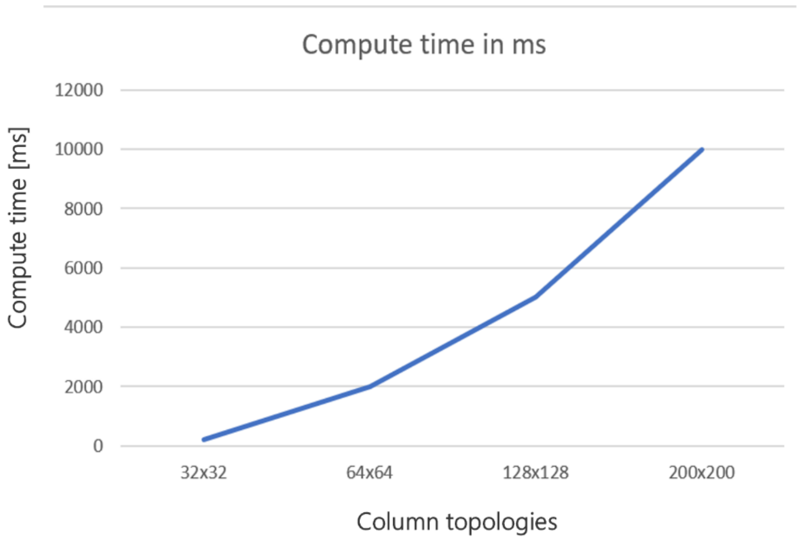


*Figure 40*
*SpatialPooler-MT computes time in milliseconds dependent on column topology.*

In a similar experiment, the SP-Parallel approach was employed instead of SP-MT, with topologies consisting of a larger number of columns and multiple nodes. The experiment aimed to measure the learning performance of MNIST images on 1, 2, and 3 nodes. As illustrated in Figure 41, it can be observed that SP-Parallel on a single node requires nearly the same amount of time as SP-MT (see Figure 40).

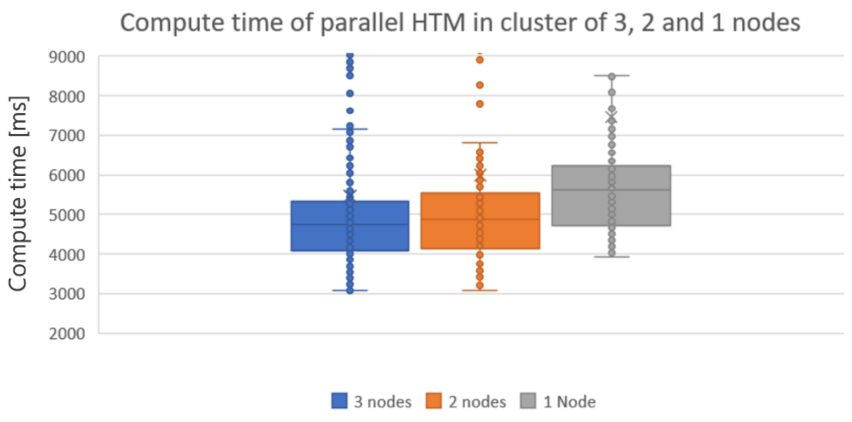


*Figure 41*

*Learning time in [ms] of an MNIST image 28x28 with 200x200 Columns.*

The obtained result indicates that the Actor model framework does not significantly consume time on internal messaging. By adding more nodes to the cluster, the performance improves as anticipated. However, the optimal number of partitions still requires further investigation. Currently, in order to enable parallel execution of calculations on multiple partitions, it is recommended to set the number of partitions to be 2 or 3 times higher than the number of cores across all nodes.

Figure 42 illustrates the memory and CPU consumption on a gathering node during distributed calculations across multiple nodes, regardless of the column topology. Both memory and CPU usage are distributed among the nodes in the cluster. The figure demonstrates that memory usage increases during the initialization phase (step 1) as space is allocated for the columns.

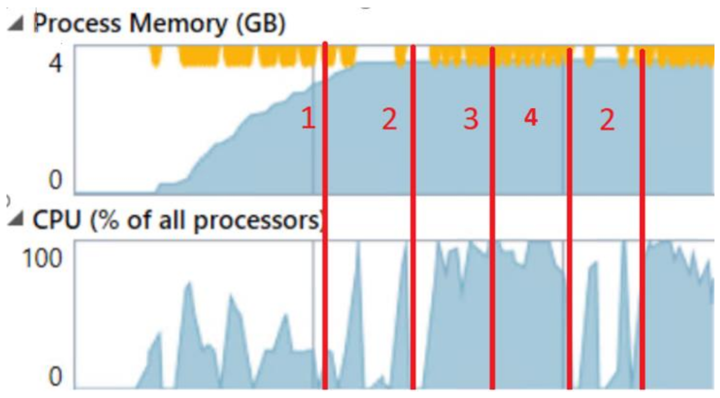


*Figure 42*

*Process memory on a node while computing of SP-Parallel is running.*

Subsequently, the memory consumption stabilizes throughout the iterative learning process during the repeating steps 2, 3, and 4. The system was then subjected to a test run with a maximum of 250,000 cortical columns. In this configuration, a total of 196,000,000 synapses were allocated to the sensory input, which consisted of 28x28 sensory neurons connected to the proximal dendrite. The experiment employed global inhibition, as discussed in section 5.7.3.1.

The final experiment, depicted in Figure 43, expands upon the previous experiment by introducing topologies that require a considerably larger number of mini-columns and synapses. Specifically, the experiment compares the performance of topologies with dimensions of 200x200, 300x300, and 500x500 (equivalent to 250,000 columns). Furthermore, it should be noted that the initialization time of the SP instance, as shown in Figure 44, should not be underestimated. The allocation of mini-columns and their corresponding synapses takes significantly longer than the actual computation time. The experiments also indicate that there is no significant difference in the compute time between the 200x200 topology with 20 partitions and the same topology with 15 partitions. This is primarily because the number of partitions exceeds the number of available cores (3 nodes with 4 logical processors) in both cases.

If the number of partitions is lower than the number of available cores, it would result in underutilization of the CPU power, while having an excessively high number of partitions would lead to frequent context switches, thereby reducing overall performance.

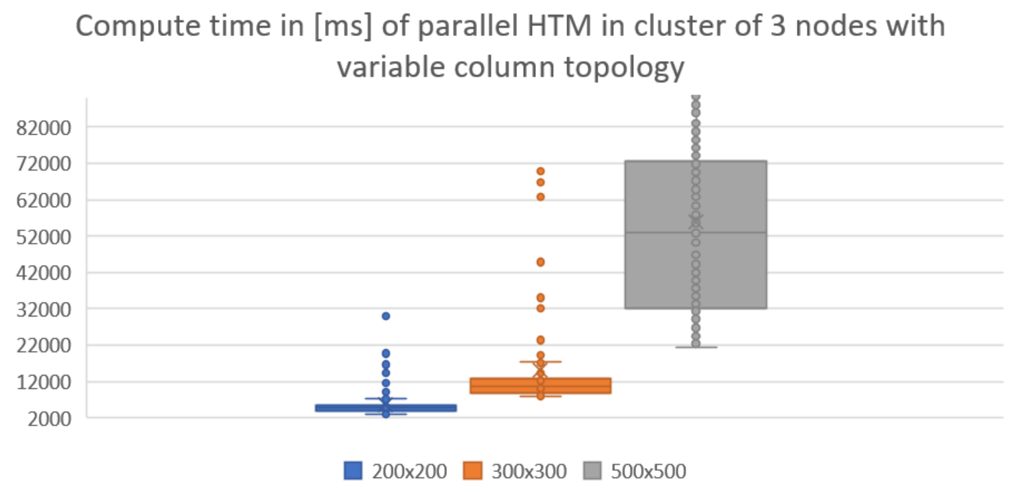


*Figure 43*
*Compute the time of SP-Parallel on three nodes depending on column topology.*

The presented results have been evaluated using an Actor model implementation, where calculations are bound to specific nodes without a persistent calculation state. The persistence of calculations typically has a negative impact on performance. Additional experiments (not included in this document) have revealed that a single column, when persisted as JSON, occupies approximately 700 kb of space. Persisting partitions, as described in this work, with thousands of columns would require gigabytes of storage and necessitate a sophisticated algorithm for efficient state saving and loading. Addressing this challenge is one of the future tasks in this research context.

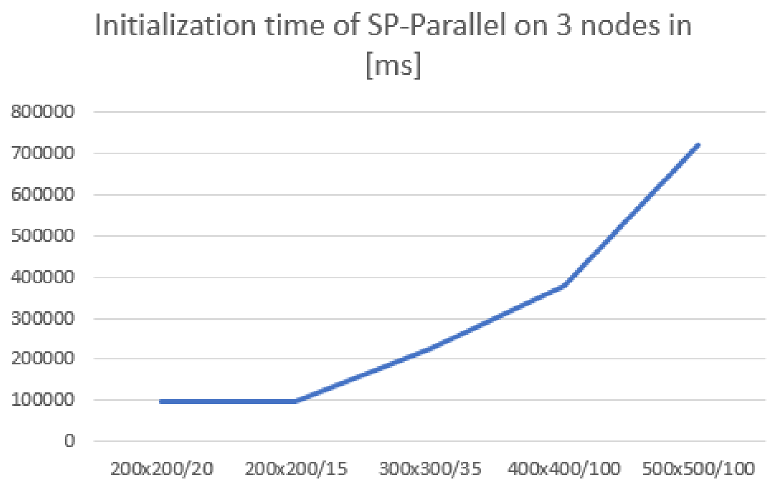


*Figure 44*
*The initialization time of SP-Parallel in milliseconds in a cluster of 3 nodes varies depending on the column topology. Topologies used in the experiment include 200x200 with 20 partitions, 200x200 with 15 partitions, and so on.*

# 7 How mini-columns develop the spatial similarity and robustness to noise

This chapter focuses on noise robustness and spatial similarity as cortical features found in SP. It describes how the sparsely encoded pattern develops robustness against noise and how the same capability detects the similarity between patterns.
The results described in this chapter were published at Symposium on Pattern Recognition and Applications, 2022, in Rome (Dobric, Pech, Ghita, Wennekers, 2020).

## 7.1 Introduction

The Spatial Pooler is responsible for clustering similar spatial patterns of active neurons into a sparse representation of mini-columns. The resulting Sparse Distributed Representation (SDR) generated by the Spatial Pooler typically occupies a small fraction, around 2%, of the total column space as defined in section 3.1.1. One of the capabilities of the SP is robustness against noise. That means the SP keeps stable output when adding noise to the input. This was initially documented in previous works (see 3.1.3). According to previous research, the trained Spatial Pooler (SP) demonstrates greater resilience against noise when compared to its untrained counterpart. The objective of this work was to examine the impact of encoding on the robustness of the learned Sparse Distributed Representation (SDR) against noise. Understanding this relationship is crucial for developing practical, real-world applications based on SP that can reliably operate with noisy inputs.
The hypothesis in this work is that the robustness against noise depends on the sparseness of the input. Based on the assumption that sparseness is defined as the fraction of active neurons divided by the total number of available neurons, the hypothesis suggests that higher sparseness leads to improved robustness.

Section 7.2 describes methods applied to measure the robustness and validate the hypothesis. Finally, experiments in Results section 7.3 proves that the higher number of non-zero bits in the input leads to more robustness against noise and better memorizing of the pattern.

This finding suggests that the input pattern can easily be boosted to memorise the reference pattern better and produce a more stable representation of SDR with a very high noise level.
The finding implies that by activating more neurons in encoding the input pattern, the ability to effectively memorize the reference pattern and generate a more stable Sparse Distributed Representation (SDR) is enhanced, even in significant noise levels.

## 7.2 Methods

To measure the robustness against noise, a reference input vector is created without and trained with ten iterations due to the fast memorizing capability of the SP. This number of iterations ensures a stable sparse representation achieved with a set of parameters shown in Table 3. Second, a portion of the noise is added to the reference input vector by flipping a specific percentage of bits (zeros bits to one-bits and vice versa). Finally, noise robustness is calculated by measuring the sensitivity of the SP to varying input noise. In this work, robustness was measured by manipulating the sparsity of the input vector using both trained and untrained Spatial Poolers (SP). To illustrate this methodology, two random images of 28x28 pixels were selected from the MNIST database (LeCun, 2010). These images were converted into binary 0-1 vectors and utilized as input during the training phase. Initially, these images were employed to evaluate the noise robustness of the trained and untrained SP. Specifically, "Image 8" exhibited a sparsity level of 17%, while "Image 7" had a sparsity level of 6%.

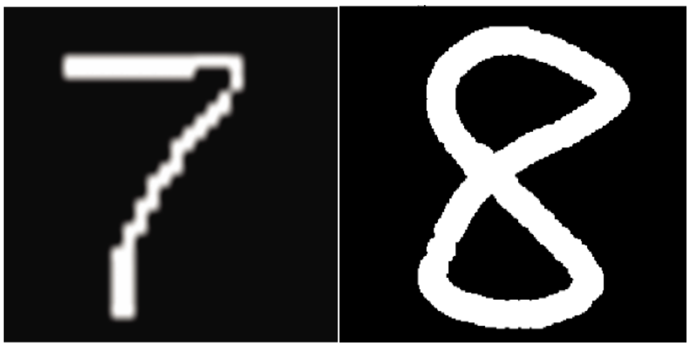


*Figure 45 - Digits "7" and "8", 28x28 pixels obtained from MNIST database.*

All experiments in this chapter were executed on the .NET core implementation of Spatial Pooler (Dobric, 2019). For every tested image, the same instance of SP was used with the configuration shown in Table 3. In these tests, Global Inhibition was used. However, Local Inhibition (5.7.3) shows nearly the same results. In addition, mini-Column Bosting (see 5.7.4) was set to a high cycle period to ensure that Spatial Pooler doesn't enter column boosting while the experiment runs[2]. Disabling boosting during experiment execution ensures that eventually, activated boosting will not clear memorized spatial patterns during the learning process. Chapter 8 briefly describes this effect and changes in the SP algorithm needed to ensure stability.

To achieve a stable Sparse Distributed Representation (SDR) of active mini-columns for both images, the learning process was configured to undergo ten iteration steps. This allowed the Spatial Pooler (SP) to reach a steady state, as described in section 5.7.3. Typically, with the configuration described in Table 3 and the given input vectors, SP attains a stable state within 2-3 steps. The choice of ten iteration steps was based on practical considerations.

As next, various levels of noise percentages, denoted as k={5, 10, 15, ..., 100}, were introduced to the reference image. For each noise level, the same instance of the Spatial Pooler (SP) was trained using the noise-distorted representation of the reference input. Following each training step, the input distance metric and output distance between the vectors were calculated.
The *distance* value introduced in this work is a metric of similarity. For example, the input vector with noise is compared with the reference input vector without noise. The same calculation of distance is done on output vectors. Finally, the distance of the output vector of the input with noise is compared with the reference output value calculated for input without noise. Subsequently, the input and output distances are compared, and their relationship is utilized as a metric to gauge the resistance against noise. The model is considered robust against noise if a lower output distance is observed for a higher input distance. In essence, the Spatial Pooler (SP) demonstrates robustness when the output remains largely unchanged despite a considerable level of noise in the input.

[2] Instability issue of the SP discussed in the next chapter. This issue was solved by introducing Homeostatic Plasticity Controller.

The distance metric in this context is a value which defines the similarity between compared vectors. First, the number of the same bits between the vector and the reference vector is calculated by Equation (47) in Algorithm 1.

$$\mathbf{o}_{\mathrm{k}} = \sum_{i=0}^{N} \parallel \mathbf{a}_0(i) \circ \mathbf{a}_{\mathrm{k}}(i) \parallel \tag{47}$$

$$\mathbf{l}_{\mathrm{k}} = \sum_{\boldsymbol{i}=\mathbf{0}}^{\boldsymbol{N}} \mathbf{a}_{\mathrm{k}}(\boldsymbol{i}) \tag{48}$$

$$\mathbf{d}_{\mathrm{k}} = \frac{\mathbf{l}_0 - (N - \mathbf{o}_{\mathrm{k}})}{\mathbf{l}_0} 100\ ;\ \mathbf{o}_{\mathrm{k}} < \mathrm{N} \tag{49}$$

Let $\mathbf{a}_0$ denotes a reference vector (input or output) without any noise and let $\mathbf{a}_{\mathrm{k}}$ represent a vector (input or output) with a noise level specified by level $k$. Both comparing vectors must have the same number of bits denoted by N. Each input vector with injected noise is bitwise compared with the reference input vector using the L0-norm described in equation (47). The equation counts the total number of non-zero elements of the noised vector $k$. The overlap $\mathbf{o}_{\mathrm{k}}$ is calculated as the number of non-zero bits at the same position. Conversely, $N - \mathbf{o}_{\mathrm{k}}$ represents the number of different bits.

In the equation (48), the value $\mathbf{l}_k$ corresponds to the number of non-zero bits in the noised vector $k$, while the value $\mathbf{l}_0$ represents the number of non-zero bits in the reference input vector without any noise. The distance $\mathbf{d}_k$ as defined in equation (49), serves as the metric utilized in this experiment to determine the similarity between vector $\mathbf{a}_{\mathrm{k}}$ and the reference vector $\mathbf{a}_0$. This metric, referred to as distance, is employed in all subsequent experiments outlined in this chapter.

| Parameters | Value |
|---|---|
| Potential Percentage | 1 |
| Local Area Density | -1 (automatically calculated) |
| Potential Radius | -1 (all inputs are connected to every column) |
| Active Mini-Columns in Area | 0,02*64*64 (2% of columns) |

|  |  |
|---|---|
| Stimulus Threshold | 0,5 |
| Synaptic Decrement | 0,01 |
| Synaptic Increment | 0,01 |
| Permanence Connected | 0,1 |

*Table 3.*
*SP parameters used in the experiment. For more detailed information about the meaning of all parameters, please see (Dobric, 2019).*

In each experiment, the predefined or randomly generated input vectors are iterated through. For each vector in the input vector list, the Spatial Pooler is trained to memorize its spatial pattern. The training process is repeated ten times to ensure that the pattern is completely learned. In this context, the pattern is considered learned if the set of active mini-columns encoded by the Spatial Pooler remains unchanged when the previously learned vector reappears.

It is important to note that no additional encoding is applied between the input vector and the Spatial Pooler (SP). Typically, when working with the SP, an encoder such as a Scalar Encoder or Date Encoder is used to preprocess the input. However, in the experiments conducted in this study, the input vectors are raw images or random vectors that are utilized directly without any additional encoding steps.

The training process was implemented using the pseudo-code presented in Algorithm 1 (Dobric, Pech, Ghita, Wennekers, 2020). Each input vector underwent training at multiple noise levels. The first noise level corresponds to zero noise, representing the reference input vector without any noise. The subsequent five levels incrementally added 5% more noise to the original (reference) input vector, resulting in noise levels of 5%, 10%, 15%, 20%, and 25%. It is important to note that the Spatial Pooler (SP) was trained using both the reference input vector and the corresponding noised input vectors at each noise level.

In each result, the metric $d_k$ is computed by comparing the output Sparse Distributed Representation (SDR) for a specific noise level with the SDR output for the same input vector without any noise (reference output).

## 7.3 Results

Figure 46 shows the *distance* of the trained Spatial Pooler for two randomly chosen MNIST images presented in Figure 45. The blue line represents the Distance Metric (similarity) between the reference input and noised input. It falls nearly linearly, as expected. The orange line represents the output distance between the SDR of noised input and the SDR of the reference input. In the case of image "7" (sparseness 6%), the output distance holds a stable set of active columns till approximately 30% of noise.

Likewise, "Image 8" (with a sparsity of 17%) remains stable until approximately 40% of the noise is introduced. These results demonstrate a certain level of noise resistance in both cases, although varying degrees. Thus, it can be inferred that "Image 8" exhibits greater noise resistance compared to "Image 7". This observation aligns with the hypothesis presented in section 7.1, suggesting that the level of resistance is influenced by the sparsity of the spatial input.

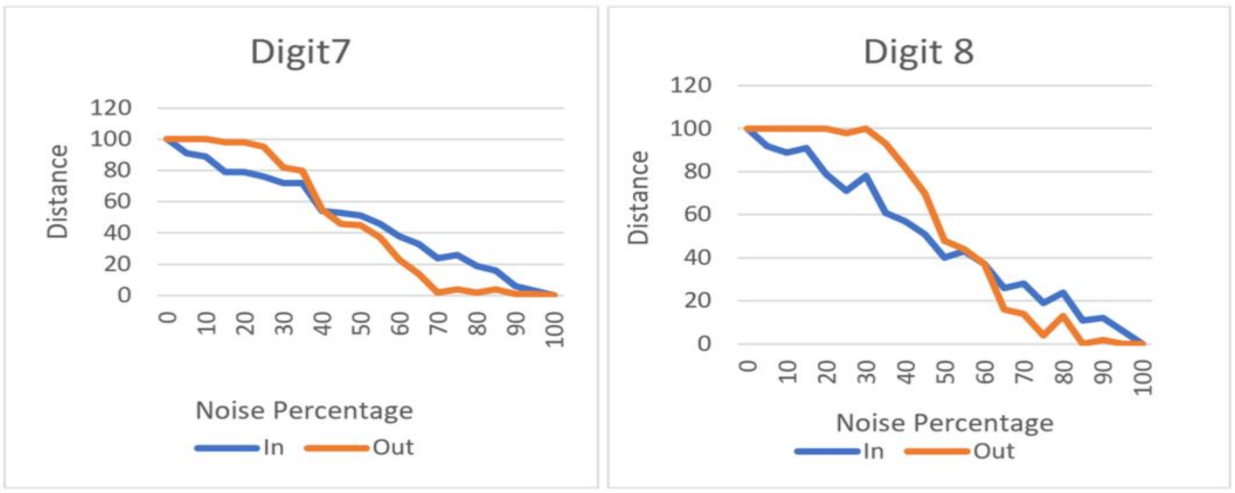


*Figure 46*

*Blue line shows calculated distance metric (similarity) between input and Input after adding noise. Orange line shows the distance between active mini-columns for the reference input and active mini-columns for encoded noised input vector.*

In order to verify this hypothesis, two box images were generated with dimensions of 1024x1024 pixels (Figure 47). The first image, referred to as "Box1", contains 240 non-zero bits, while the second image, "Box2", contains 110 non-zero bits. This results in sparsity levels of 0.24 and 0.11, respectively, indicating that "Box2" is sparser than "Box1". In the experiment, noise levels ranging from 5 to 100 percent were added to both reference images.

Figure 48 shows the encoding of the five noise levels for each of the two boxes. Then, the SP was trained with both reference and noised images. Every image in the figure on the left represents the input used for Spatial Pooler.

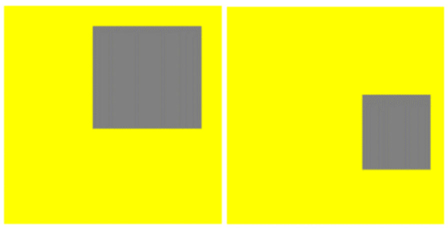


*Figure 47*

*For training process, two boxes with dimensions of 1024x1024 pixels were utilized. The left box had a sparsity of 0.24, while the right box had a sparsity of 0.11. These boxes were visually represented in yellow color. A comparison was made between the two images, taking into account their different numbers of non-zero bits.*

The resulting encoded Sparse Distributed Representation (SDR) is presented as a sparse output of active mini-columns, providing a top view of the representation. The SDR output is displayed for various input scenarios, including the reference input without noise (0%), as well as inputs with 5%, 10%, 15%, and 20% noise levels. Referring to Figure 49, it can be observed that Box 1 is still accurately recognized, with an output distance of approximately 100%, given an input distance of around 40% (as shown in Figure 47 on the left).

The results for Box 2 exhibit some variation. The image containing Box 2 is correctly recognized for inputs with noise levels up to 25%. This implies that Box 1, with a greater number of nonzero bits, demonstrates superior noise robustness compared to Box 2. This outcome serves as a compelling indication that the robustness is indeed influenced by the sparsity of the input. To further validate this assumption, numerous random input vectors were employed, with the percentages of non-zero bits being varied at a specified noise level.

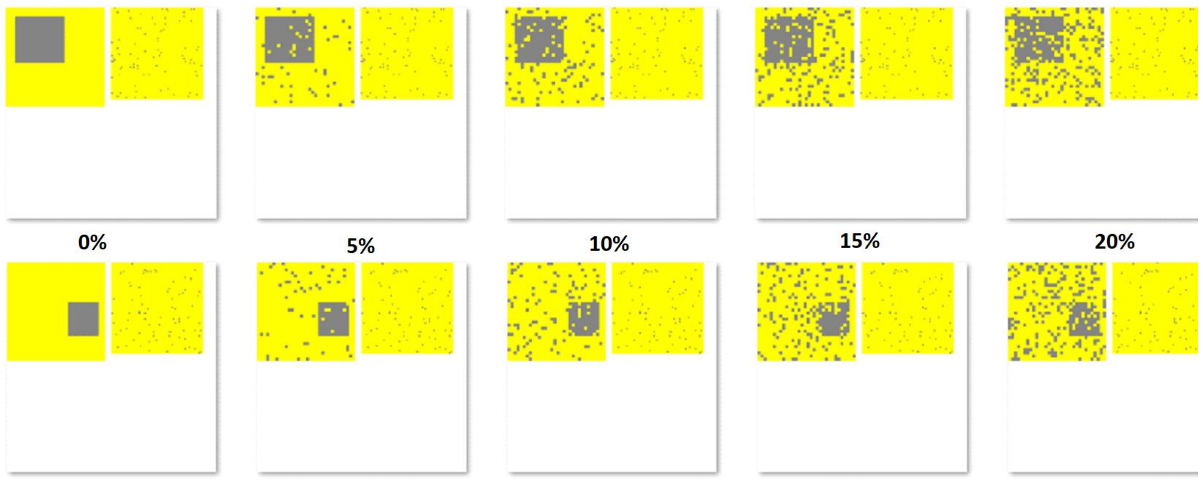
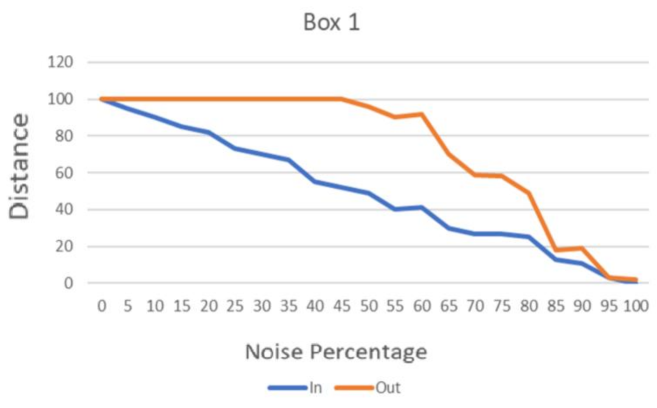


*Figure 48*

*Images on top show the noised input and output of Box1. Images on the bottom show the noised input and output of Box 2. Each image represents the input vector with a specific noise level (on the left) and their sparse representation of active columns (on the right).*

The results of the experiment are shown in Figure 50. Note that Figure 50 has a slightly different representation than Figure 49. Instead of distance and noise, the input and output overlap are compared in Figure 50. The more noise is contained in the input vector, the less overlap between noised and input reference vector. For the same noisy input vector, the output overlap is calculated between the output SDR of the noised vector and the reference output SDR encoded by the reference input without noise.



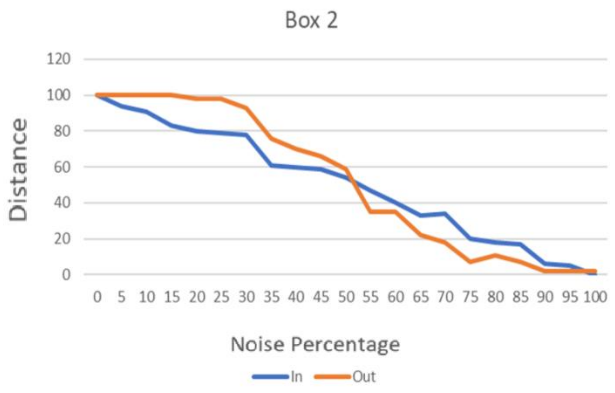


*Figure 49*

*Noise resistance of two boxes. The blue line shows the distance between noised input and reference input vectors without noise. The orange line shows the same distance between the output of the trained Spatial Pooler of noised input and the output of the reference input.*

Upon comparing the results between enabled learning (Figure 50 - left) and disabled learning (Figure 50 - right), it can be inferred that the Spatial Pooler maintains a stable SDR output when learning is enabled. However, the Spatial Pooler with disabled learning does not memorize the pattern and is not resistant to noise. As shown in the figures on the right for all experiments, the output distance changes even faster without learning enabled than the input distance, especially for lower noise levels. Also, the Spatial Pooler does not show good resistance with enabled learning by using less than 10% of non-zero bits (first figure on the left). In this case, even slightly deforming the input immediately deforms the output, which is not the desired result.

Nevertheless, when learning is enabled and the input contains a greater number of non-zero bits, the distance between the output of the noised vector and the reference output, which has zero noise input, remains nearly constant across a relatively broad range of noise levels. This observation suggests that the Spatial Pooler retains the memory of the reference value even when exposed to highly noisy conditions.
For instance, when the number of non-zero bits is increased to 20% (as shown in the

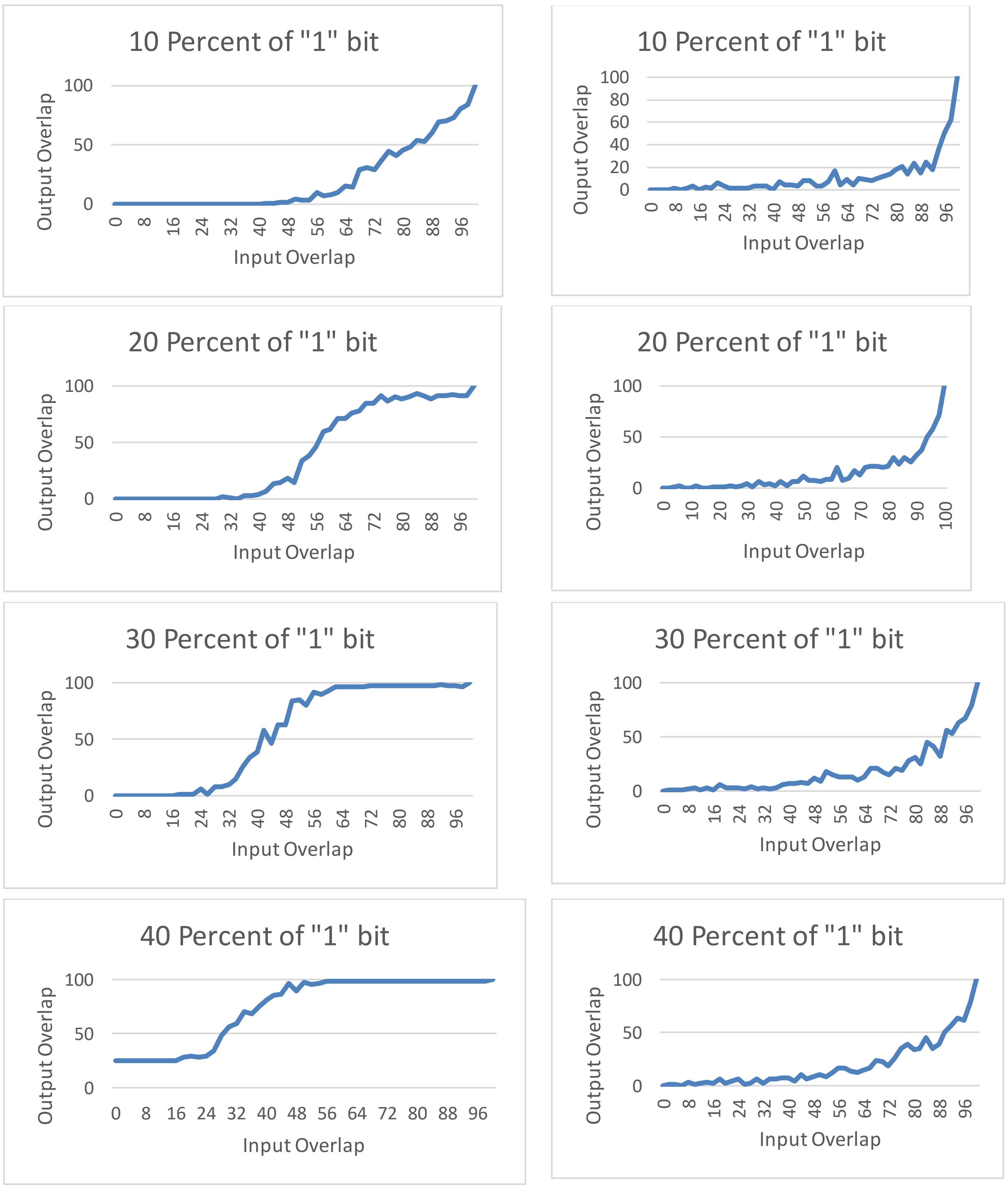


*Figure 50*

*The impact of the number of non-zero bits on the output overlap is depicted in the graphs, with training enabled (on the left) and training disabled (on the right). The X-axis indicates the input overlap as a percentage, while the Y-axis represents the output overlap.*

second figure on the left), the output remains stable with an output distance of over 90% even when subjected to approximately 25% input noise. Similarly, with 30% non-zero bits, the Spatial Pooler maintains nearly unchanged output even with up to 50% distorted input. This serves as a strong indication of the robustness of the Spatial Pooler. The experiment demonstrates that a higher number of non-zero bits grouped together, forming a certain pattern in the input, leads to increased robustness. However, it is worth noting that a higher number of non-zero bits reduces sparsity and diminishes the capacity of the Cortical Algorithm.

Finally, the same experiment has been executed for a large set of random input patterns. The experiment's goal was to investigate the behaviour of robustness to noise in the case of random patterns.

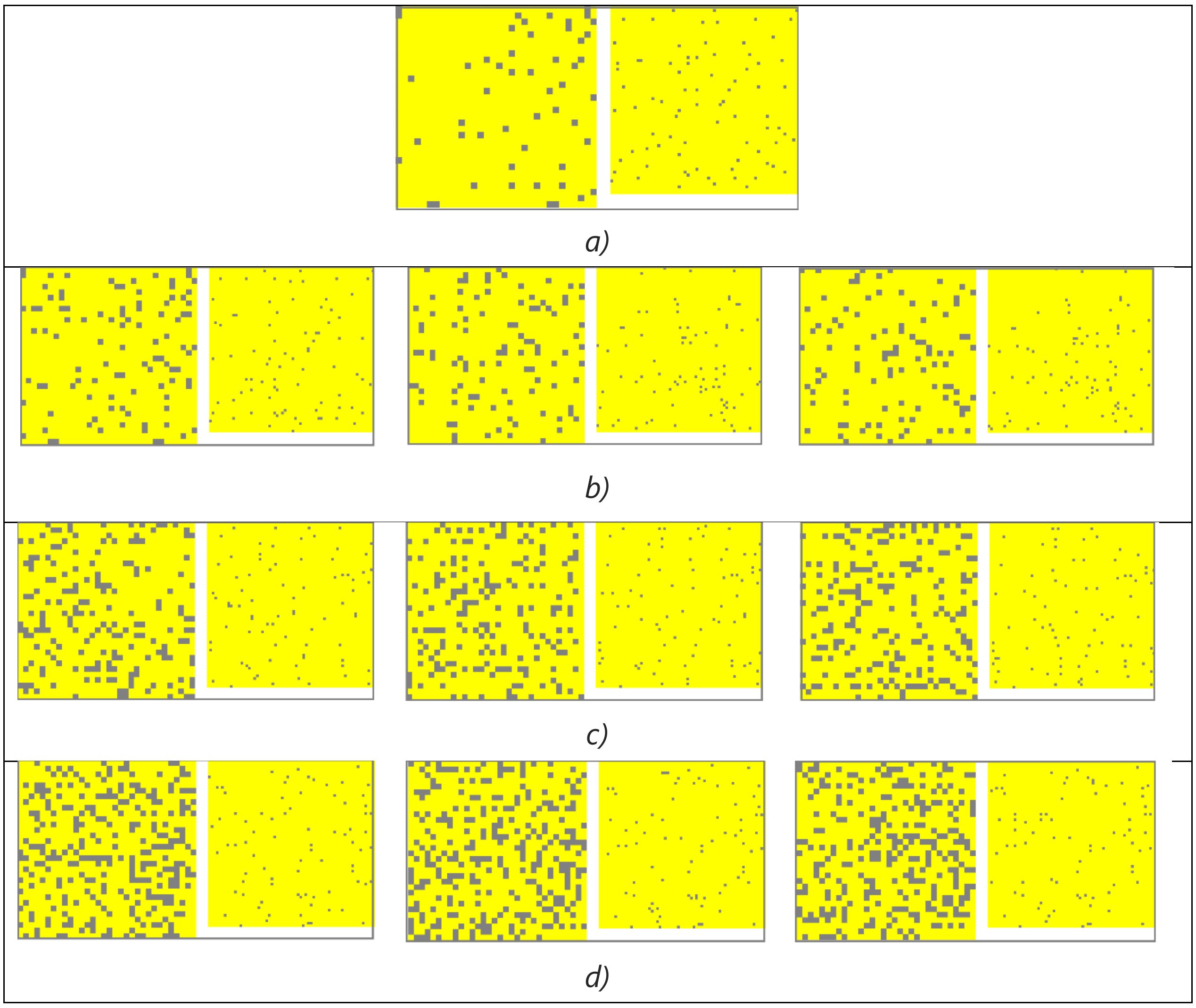


*Figure 51 Reference Random Vector with three examples of 5, 10 and 15 percent noise levels.*

In this case, the input is generated to distribute active bits with a given noise percentage uniformly. Figure 51 shows the reference Random Vector (a) along with three random vectors with noise levels of 5% (b), 10% (c), and 15% (d).

The noise was generated randomly by flipping zero bits to one and vice versa at random positions, starting with the reference vector. As previously described, the left side of each image represents the input shown to the SP, and the right side shows the corresponding SDR. Figure 52 shows the distribution of the output distance for random vectors with 5,10,15, and 20 percent of the noise.

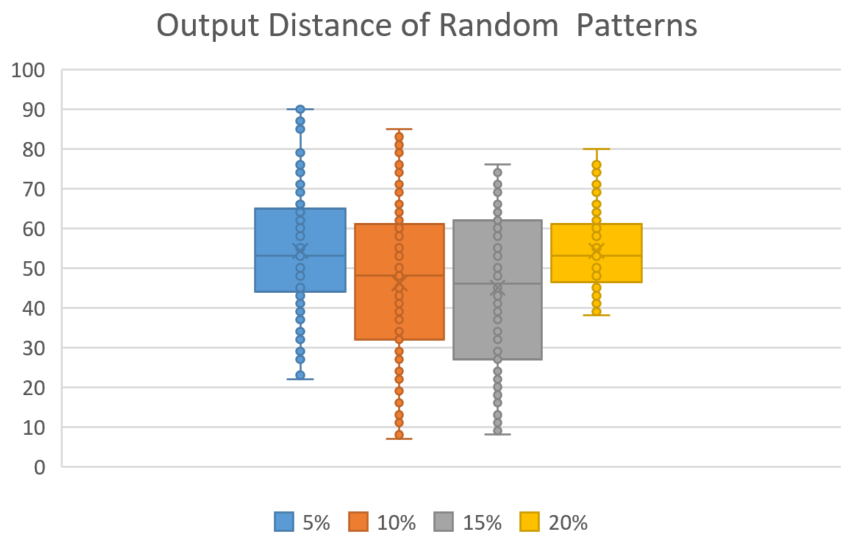


*Figure 52 Distribution of output distance when presenting random input patterns*

The results demonstrate that the Spatial Pooler does not retain random patterns when even a tiny portion of noise (i.e., 5%) is added to the reference vector. It can be concluded that the SP is not robust to noise in the case of random input patterns, as shown in the previous figure. Robustness is achieved with patterns formed by grouping non-zero bits together.

# 8 Homeostatic Plasticity Control and the Need for the newborn stage

This chapter describes how spatial learning in the SP was improved by introducing a *newborn* stage of the cortical algorithm. It demonstrates the importance of the stage that the cortical algorithm starts at the beginning of learning and needs to activate all areas with the help of the homeostatic plasticity mechanism.
Inspired by findings in neurosciences, it shows how plasticity enhances the learning process and clarifies why a *newborn* stage is essential for the cortical algorithm of species. Furthermore, the extended algorithm prevents the original SP algorithm from entering unstable ("epileptic") behaviour.

This part of the work related to mini-column plasticity was first published in 2021 at the International Conference on Pattern Recognition Applications and Methods (ICPRAM) in Vienna and awarded as the best industrial paper (Dobric, Pech, Ghita, Wennekers, 2021).

Further research in this domain builds upon the previous study and addresses the remaining instabilities. It specifically investigates the plasticity of synapses and demonstrates how the Spatial Pooler (SP) can achieve complete stability by controlling and disabling the excitation of inactive synaptic connections between input neurons and mini-columns, particularly after the newborn stage. Results of extended work were published in 2022 in the Springer Nature Computer Science Journal (Dobric, Pech, Ghita, Wennekers, 2022).

## 8.1 Introduction

Experiments in this work show an unusual behaviour of the SP, which indicates that the SP, by design, can become unstable over time. That means that learned patterns will be forgotten and learned again during the learning process. This is known as the stability-plasticity dilemma (see Chapter 3). The learning and forgetting repeat continuously over time, meaning that the Spatial Pooler oscillates between stable and unstable behaviour. Moreover, experiments also show that instability is related to specific patterns and not necessarily to the entire set of input patterns.

For example, The SP can hold the stable $SDR_1$ for a specific pattern *i1* while $SDR_2$ for another pattern *i2* becomes unstable. The stability of the SP is crucial for applications that depend on spatial pattern recognition. Since the Sparse Distributed

Representations (SDRs) generated by the SP are utilized as input for the Neural Association Algorithm, an unstable SP would adversely affect the performance of subsequent algorithms. In this part of the research, the instability of the SP was investigated and compared with biological observations. As a result, a modified version of the SP was developed and tested to address this instability. The extended algorithm not only ensures the stability of the SP but also prevents it from exhibiting unstable or "epileptic" behaviour, as observed in the original algorithm. This chapter describes how and why the Spatial Pooler forgets learned SDRs during training. It also introduces the *newborn* stage of an artificial algorithm like SP.

The concept of the *newborn* stage in the modified algorithm was inspired by neuroscientific research (Maffei, Nelson, Turrigiano, 2004), which indicates that a plasticity mechanism is predominantly active during the early development of newborn mammals and subsequently deactivated or shifted away from the cortical layer L4, where the Spatial Pooler (SP) operates. This insight guided the design of the modified algorithm, ensuring that the SP behaves in a manner consistent with the biological observations and developmental processes in mammals.

As explained in the following section, the new algorithm controls the boosting of inactive mini-columns and weak synapses. The new SP with the *newborn* stage was designed first to enable the homeostatic plasticity mechanism (Turrigiano, Nelson, 2004) that boosts <u>inactive mini-columns</u> and exitize <u>weak synapses</u> and then disables them (see 5.7.4). The final solution of the extended SP clearly shows that learned SDRs remain stable during the lifetime of the Spatial Pooler.

The extended algorithm introduces a mechanism to regulate the activation and strengthening of <u>inactive mini-columns</u> and <u>weak synapses</u> within the Spatial Pooler (SP). This mechanism, inspired by the concept of homeostatic plasticity (Turrigiano, Nelson, 2004), initially enables the boosting of these inactive elements during the newborn stage and subsequently disables them (as described in section 5.7.4). Through this approach, the modified SP achieves stable learned Sparse Distributed Representations (SDRs) that persist over the lifespan of the algorithm. The final solution demonstrates the effectiveness of the extended SP in maintaining stability and robustness in pattern recognition tasks.

## 8.2 Methods

To analyze the learning process of the Spatial Pooler, a specific instance of the SP was utilized, employing a set of common parameters outlined in Table 3. The experiments conducted in this work involved varying the number of mini-columns, and the results presented here were obtained using 2048 mini-columns. As described in section 5.5, the scalar encoder was employed to encode the scalar input values used to train the SP. The SP was trained to encode input ranging from 0 to 100. Prior to presenting an input to the Spatial Pooler, each input value was encoded using 200 bits, with 15 non-zero bits representing each value.

In
Figure *53*, three examples of encoded scalar values (0, 1, and 2) are depicted, which were utilized as input for the SP in this experiment. For a comprehensive understanding of the significance of all parameters, please refer to (Dobric, 2018). The encoded input value is displayed on the right side, while the corresponding SDR is shown on the left. In the figure, the grey colour represents zero bits, representing the background of the image, while the black colour represents active bits. The grey dots on the left represent active mini-columns after the input has been encoded to SDR.

As mentioned in section 2.6, the Spatial Pooler implements a boosting of inactive mini-columns and excitation of passive (weak) synapses. Moreover, the plasticity mechanism is implemented inside the Spatial Pooler (Damir Dobric, Andreas Pech, Bogdan Ghita, Thomas Wennekers, 2021). This mechanism guarantees the uniform utilization of all mini-columns across all observed patterns (refer to 5.7.4.2).

The Spatial Pooler converts each input pattern into a Sparse SDR, which is expressed as a collection of active mini-column indices, denoted as $A_k$ for the given pattern in iteration k (see 5.7.2). Then, in every learning step of the same pattern, the similarity between SDR in step k and step k+1 is calculated as shown in equation (50).

$$s = \frac{|A_k \cap A_{k+1}|}{\max\left(|A_k|, |A_{k+1}|\right)} \qquad (50)$$

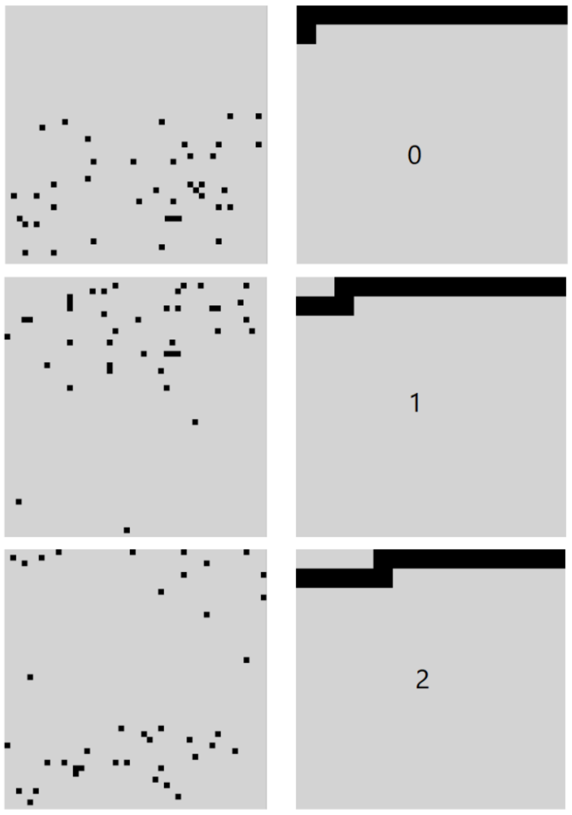


*Figure 53:*
*The Scalar Encoder (on the right) encodes three input values, while the corresponding Sparse Distributed Representation (on the left) is encoded by the Spatial Pooler. The figure shows examples of the encoded values 0,1, and 2.*

According to the equation, the similarity, denoted as $s$, is a ratio that compares the number of identical active mini-columns in SDRs generated at steps $k$ and $k+1$ to the maximum number of active mini-columns observed in two comparison steps. The SP is considered stable if the generated SDR for a particular pattern remains unchanged throughout its entire lifespan. In such cases, the similarity between all SDRs corresponding to the same pattern is 100%.

In Figure 54, the SDR of a specific input pattern presented to the SP is displayed over 25000 iterations. The SP demonstrates rapid learning, typically requiring only a few iterations to learn presented input. The y-axis represents the similarity, denoted as $s$, between the SDRs of the current and previous iteration steps. The x-axis indicates the iteration step number. A similarity of 100% means that the learned SDR remains consistent over time. However, after an unknown number of learning steps, the SP forgets the already learned SDRs and begins learning anew. As a result, the similarity drops from 100% to zero or another value.

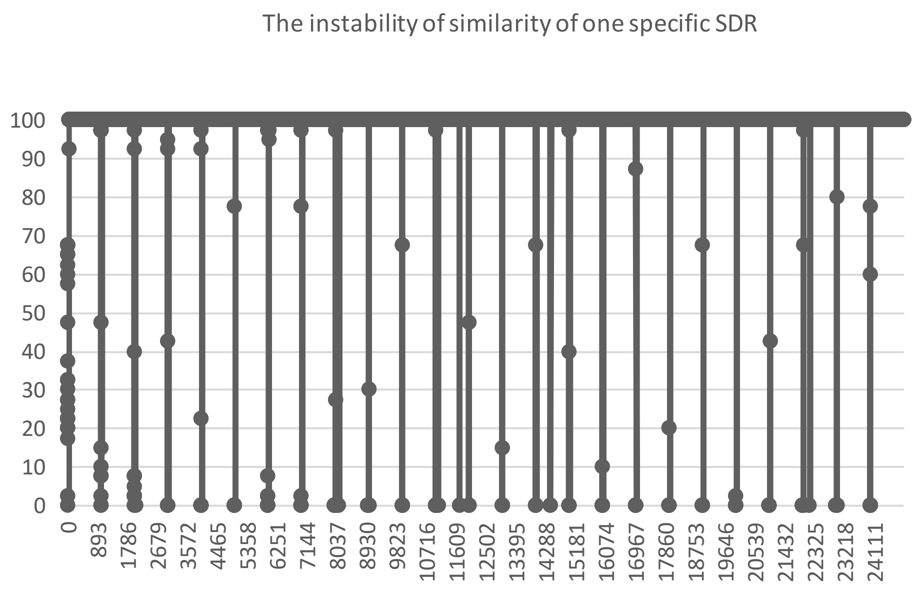


*Figure 54*

*The graph illustrates an unstable Spatial Pooler (SP). The y-axis represents the similarity of the SDR for a specific pattern in each iteration step (x-axis). Initially, the SP learns the pattern over 25000 iterations, and during this time, the SDR remains unchanged. After an unspecified number of iterations, when boosting becomes active, the SP forgets the previously learned SDR, resulting in a drop in similarity. Subsequently, the SP begins the learning process again.active.*

Contrarily, maintaining a similarity of 100% implies that the learned SDR remains constant for the entire duration of the iteration. A similarity below 100% indicates that the generated SDRs for the same input differ, revealing an unstable Spatial Pooler. As shown in Figure 54, the learning state oscillates between stable and unstable states throughout the learning process, which is impractical for applications. This experiment demonstrates the instability of the Spatial Pooler but does not delve into the specific encoding details that cause the unstable SDR. Figure 55 presents the same behaviour, illustrating the encoding of the SDR for the same pattern in the first 300 iterations (cycles) using a single input value.

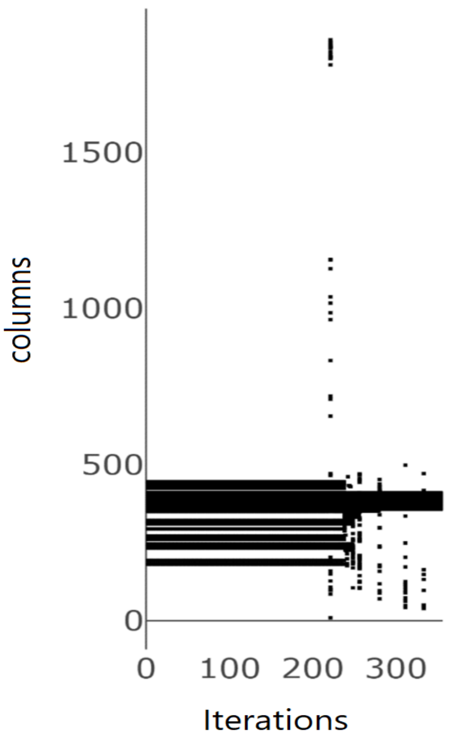


*Figure 55*
*In the first approximately 350 iterations, the chosen input is encoded to SDR.The encoded SDR remains stable for approximately the first 220 iterations. After this point, the SDR undergoes changes, leading to instability in subsequent iterations.*

Initially, the Spatial Pooler produces a stable SDR during the early stages of the learning process and maintains its stability (unchanged) for around 200 iterations. However, subsequent iterations result in unstable SDRs, followed by a return to the stable state (not depicted in the figure), and this pattern continues intermittently.

In the conducted experiment, the boosting of inactive mini-columns was deactivated by assigning zero values to the parameters DUTY CYCLE PERIOD and MAX BOOST. These parameters regulate the frequency of mini-column activation (refer to section see 5.7.4.2). Disabling boosting in the Spatial Pooler (SP) leads to stable SDR generation, as demonstrated in Figure 56. The figure illustrates a specific instance of a stable encoding for a single pattern, achieved by employing the disabled boosting algorithm. During the initial iterations, the SP learns the pattern and successfully encodes it into a stable SDR.

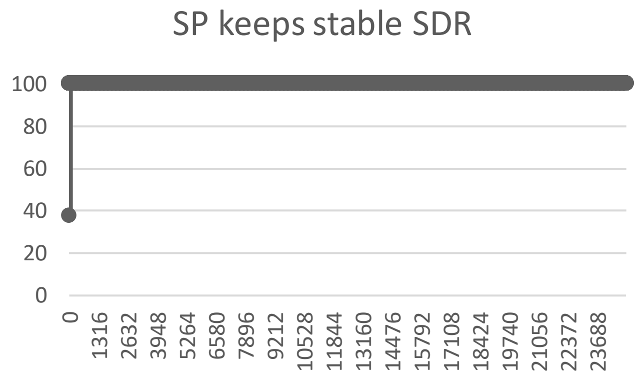


*Figure 56*
*Spatial Pooler generates stable SDR after the boosting is disabled*.

Based on the findings, it can be concluded that turning off the boosting algorithm can result in a stable Spatial Pooler (SP). However, this approach leads to SDRs with varying numbers of active mini-columns since the uniform activation of all mini-columns is not guaranteed (see 5.7.4.2). Figure 57 shows the SDRs of two input values ‘0’ and ‘6’. With disabled boosting, the SP will also enter a stable state for all inputsDuring this experiment, the value '0' was encoded with approximately 40 active mini-columns, while the value '6' was encoded with only 4 active mini-columns. This is a significant unwanted difference. Experiments showed (not presented in this work) that some patterns can even be encoded without any active mini-column if boosting is ultimately disabled or disabled early. The SP needs time to activate all mini-columns in the cortical area. This time interval measured in a number of iterations plays an essential role in this work.

When there is a substantial discrepancy in the number of active mini-columns within SDRs for different inputs, the subsequent processing of memorized SDRs can become inefficient. This inefficiency arises because many operations in a cortical algorithm depend on calculating the overlap function (see 5.7.1). In that case, SDR-s with a significantly higher number of active mini-columns will have a higher probability than patterns with fewer ones. This unbalance will cause the generation of false positives inside the algorithm. As previously discussed, the homeostatic plasticity keeps all mini-columns in balance, so the SP should do the same.

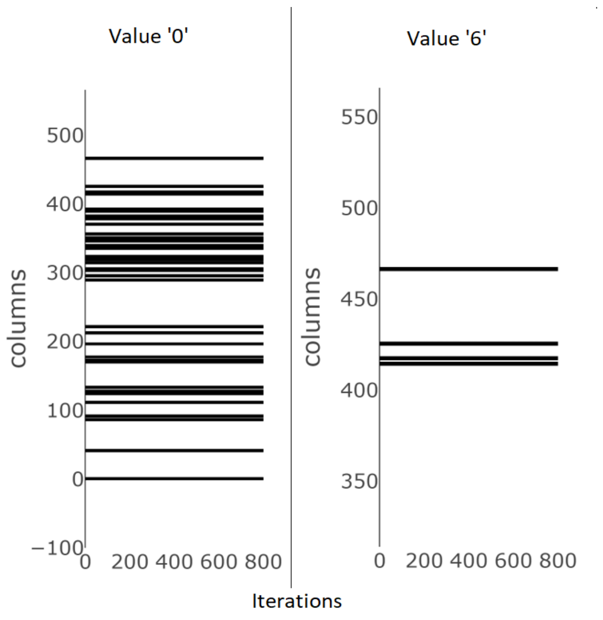


*Figure 57*

*The Spatial Pooler, with the boosting algorithm disabled, generates two SDRs with varying counts of active mini-columns.*

The parameter NUM ACTIVE COLUMNS PER INH AREA sets the percentage of mini-columns within the inhibition area that will be used to encode a pattern. As explained in section 3.1.1, this value is typically set to 2%. In the conducted experiments, with global inhibition (see 5.7.3) utilizing 2048 mini-columns, the SP generates SDRs containing 40 active mini-columns. However, when boosting algorithm described in 5.7.4.2 is activated, passive mini-columns are subsequently raised. This alters the synaptic permanence values learned by the SP, as outlined in section 5.7.2. and leads to forgetting learned patterns (SDRs), and learning resumes when the same pattern reoccurs.

In summary, disabling boosting allows the SP to achieve a stable state. However, it results in SDRs with varying numbers of active mini-columns for different inputs. On the other hand, enabling boosting ensures uniform activation of mini-columns, but it leads to unstable learning, which is the default behaviour of the SP.

The findings in neural sciences, as mentioned in section 2.6, indicate that homeostatic plasticity is primarily active during the early stages of development in newborn mammals. Building upon this insight, this research extends the Spatial Pooler algorithm by incorporating the concept of a newborn stage within the Spatial Pooler.

The deactivation of boosting homeostatic plasticity observed in the cortical layer L4 can also be implemented in the SP. Although the precise workings of this mechanism in the brain are not yet fully understood, a similar approach can be adopted within the SP. Currently, this mechanism involves boosting inactive mini-columns and boosting (exciting) mini-columns with a relatively low overlap compared to others (see 5.7.4.2).

The idea in this part of the work, aiming to stabilize the SP and keep using the plasticity, is to add an algorithm to SP that does not entirely change the existing one. The extension of the SP algorithm (Dobric, 2021) is implemented in the Homeostatic Plasticity Controller (HPC) component. The newborn stage is achieved by attaching the HPC to the existing implementation of the SP. Following the computation in each iteration, both the input pattern and its corresponding SDR are transmitted from the SP to the HPC. The HPC keeps the boosting active until the SP enters the stable state, measured over the given number of iterations.

During this period, the SP operates in what is referred to as the *newborn* stage, producing outcomes similar to those shown in Figure 54 and Figure 55. Once the SP reaches a stable state, the HPC will deactivate the boosting mechanism and inform the application of the state change. The controller monitors the involvement of mini-columns across all observed patterns. When it observes that the mini-columns are evenly utilized and that the SDRs generated for the observed patterns consist of roughly the same number of active mini-columns, the SP is considered to have entered the stable state by default. At this point, the SP exits the *newborn* stage and resumes its regular operation without employing the plasticity algorithms described in sections 5.7.4.1 and 5.7.4.2. These plasticity algorithms are deactivated by setting the configuration values *MinPctOverlapDutyCycles*, *MinPctActiveDutyCycles*, and *MaxBoost* to zero.

Switching off these values initiates the exit of the *newborn* stage of the Spatial Pooler. From that moment, the HPC will count the stable cycles and enter the stable state if the required minimum of stable cycles (`numOfCyclesToWaitOnChange`) is reached. The following section describes the HPC (Dobric, 2021) algorithm in more detail.

## 8.3 Results

The objective of the following experiment was to demonstrate the efficacy of utilizing the Homeostatic Plasticity Controller in improving the SP algorithm's ability to consistently achieve a stable state. The experiment, outlined in Listing 4, involved running 25,000 iterations and presenting 100 scalar values to the SP. The experiment was repeated numerous times, employing different configurations. The scalar encoder, specified in line 11, utilized a set of parameters (line 5) as described in Listing 4. Each input value within the range of 0 to 100 was encoded as a 200-bit vector. Furthermore, every value within the specified range was encoded using 15 non-zero bits, as illustrated in Figure 51 (right).

Listing 4. Using improved SP - Pseudo code

```
00 | function Experiment( inputPatterns )
01 | begin ( I )
02 |   p          // SP configuration parameters.
03 |  hp,enp   // HPC and encoder configuration parameters
06 |  isStable = false
05 |  en←create(enp)
06 |  hpc ←create(hp, onStateChange);
07 |  sp ←create(i, hpc);
08 |  FOR  i = 0;  i  <  25000
09 |     FOREACH  i  IN inputPatterns
10 |         // Learn and generate SDR for the input.
11 |         o ← sp.compute(encode(i));
12 |         IF isStable = true
13 |            // newborn stage exited
14 |            // Use stable SDRs. Custom code here.
15 |         ENDIF
16 |     ENDFOREACH
17 |  ENDFOR
18 | end
19 |
20 | function onStateChange(state)
21 |  begin
22 |      isStable = state // The state has changed. SP becomes stable or unstable.
23 |  end
```

A new instance of the SP is instantiated with a predefined set of parameters (line 3). This configuration is consistent with the experiment described in the preceding section, which yielded the results displayed in Figure 54 and Figure 55. The HPC is commonly associated with the SP instance (line 7, second argument) and utilized within the compute method. To facilitate practical implementation, the HPC offers a callback function (line 6, second argument) that is invoked when the controller identifies the SP has reached a stable state. The experiment is structured to execute a specified number of training iterations (line 8 sets the value to 25000). In each iteration, the SP undergoes training utilizing the complete set of input values denoted as **I** (line 9).

The spatial input undergoes training in line 11, resulting in an SDR code (a set of active mini-columns) represented by the output variable **o**. Prior to being presented to the SP, the input value i is encoded using the Scalar Encoder. This encoder is configured with a specific set of parameters described in Table 4. The Scalar Encoder is implemented as a function e, which transforms the given scalar value into a binary array.

$$e: \mathbb{R} \longrightarrow \{0,1\}$$

The computation within the SP operates solely on binary arrays, resembling the functioning of the neocortex (refer to section 3.1). In this work, the SP compute algorithm has been expanded to include the activation of the Homeostatic Plasticity Controller (HPC) outlined in Algorithm 17. The HPC computation takes place after the Spatial Pooler completes the current iteration.

| Parameters | Value |
|---|---|
| W – number of required active bits to encode a single value | 15 |
| N – Total number of input bits | 200 |
| MinVal – Minimum encoded value | 0 |
| MaxVal – Maximum encoded value | 100 |

*Table 4*
*Scalar Encoder parameters*

The HPC Algorithm takes two inputs: the binary array representing an encoded input pattern and the SDR calculated by the SP for that input. Initially, the algorithm does not make any changes to the SP, entering a phase referred to as the newborn stage. Once the minimum required number of iterations (***m*** or minCycles) is surpassed (line 15), the HPC disables boosting in the SP. For all subsequent iterations exceeding ***m***, the boosting of inactive mini-columns (Algorithm 8) and mini-columns with low overlap (Algorithm 7) is turned off by setting the parameters *MinPctActiveDutyCycles*, *MaxBoost*, and *MinPctOverlapDutyCycles* to zero.

The first two parameters, *MinPctActiveDutyCycles* and *MinPctOverlapDutyCycles*, are responsible for updating the boost factors for each column during every iteration. In Algorithm 7, these boost factors are utilized in the SP to enhance the synapses of inactive mini-columns, increasing the likelihood of their activation. By setting *MinPctActiveDutyCycles* to zero, the condition in line 13 of Algorithm 8 is not met, preventing the boosting of permanence values. Similarly, when *MinPctOverlapDutyCycles* is set to zero, the code in lines 18-21 of Algorithm 7 is not executed, thereby eliminating the boosting of synapses with low overlap frequency (weak synapses).

Algorithm 17. The Homeostatic Plasticity Algorithm

```
01 | input:   $\boldsymbol{i}$ // The sensory input.
02 | output: $\boldsymbol{o}$ // Set of active mini-columns that defined the SDR.
03 | configuration parameters:
04 |       b // SP max boost
05 |       d // SP min pct. overlap duty cycles
06 | begin
07 |   H ← $hash(\boldsymbol{i})$; // Calculate the hash value of the input of N bits.
08 |   E ← $(\mathrm{H}, \sum_{k=0}^{M} o_k)\ o_k \in \boldsymbol{o}$ // Calculate the number of active mini-columns in SDR.
   |   // The average change of num. of the act. Columns in p steps.
09|   $\delta \leftarrow \frac{1}{p} * \sum_{k=0}^{p-1} | e_{Hk} - e_{H(k+1)} | \ e_H \in \mathrm{E}$
  |   // Calculate the correlation between the current and the previous output.
10|   $\boldsymbol{c} = corr(\boldsymbol{o}', \boldsymbol{o}) \mid \boldsymbol{o}' \in \mathcal{H}$
  |   // Store input-hash and SDR pair
11|   $\mathcal{H} \leftarrow (\mathrm{H}, \mathrm{o})$
  |   // Increment the counter of stable iterations for i.
12|   $\Gamma \leftarrow \gamma_H + 1[\delta = 0, \boldsymbol{c} > \theta | 0.9 < \theta < 1, \gamma_H \in \Gamma]$
  |   // Fire stable state event
13|   $StableState[\gamma_H = \tau, \forall \gamma_H \in \Gamma, \tau \in \mathbf{N}]$
  |   // Reset the counter of stable iterations for $\boldsymbol{i}$.
14|   $\Gamma \leftarrow 0_H[\boldsymbol{c} \leq \theta | 0.9 < \theta < 1.0]$
  |   // Disable boost and synaptic excitation after specified num. of iterations.
15|   $inactiveColumnBoost = \boldsymbol{off}; synapseExitation = \boldsymbol{off}[iteration >= m]$
16| end
```

Upon disabling plasticity, the algorithm initiates the tracking of all encountered patterns and their corresponding SDRs. To avoid storing the entire input dataset internally, the hash function (line 6) computes a hash value over the sequence of input bits for the current iteration. This computed hash value is represented as a sequence of bytes and defined as the set H (line 7). Next, in line 8, a tuple consisting of the input's

hash value H and the number of active columns in the associated SDR is linked with the set $\mathrm{E}$. The set $\mathrm{E}$ retains p tuples of this information.
As mentioned, the objective is to maintain a constant number of active mini-columns across the entire SDR space. The value δ denotes the average change in the number of active mini-columns within each interval of p SDRs (line 9). The cycle interval p refers to the number of previous iterations utilized for calculating δ. In most experiments, this value was set to five, which is deemed appropriate because once the Spatial Pooler enters a stable state, the number of active mini-columns remains constant. Therefore, during the SP's unstable state, a five-state interval is sufficiently long to compute δ as a non-zero value.

The value $\delta$ is calculated as an average sum of deltas $\mathrm{e}_{Hk} - \mathrm{e}_{H(k+1)}$ in the last $p$ iterations for the given input hash value H.

$$\delta = \frac{1}{p} * \sum_{k=0}^{\mathrm{p}-1} | e_{Hk} - e_{H(k+1)} | \; e_H \in \mathrm{E} \qquad (51)$$

The stability of the new Spatial Pooler depends on two conditions. The first condition is satisfied when the value $\delta$ is zero, indicating that the total count of active mini-columns in the SDR for a particular input remains unchanged over a certain number of iterations (p). The second condition for the stability of the Spatial Pooler is to achieve a consistent SDR for every input encountered throughout the entire training process.

To ensure this, the set $\mathcal{H}$ is utilized to store tuples (H, o) consisting of input hash values and their corresponding SDRs. In subsequent iterations, the SDRs of inputs replace the previously stored tuple of the current input, thereby updating the information in $\mathcal{H}$.
$\mathcal{H}$ contains a unique tuple (H, o) for each input. These tuples in $\mathcal{H}$ are employed to compute the correlation $\boldsymbol{c}$ between the previous and current SDRs of the specific input, as indicated in lines 10 and 11.

If the correlation between the last SDR $\boldsymbol{o}'$ and the new (current) SDR $\boldsymbol{o}$ of the given input $\boldsymbol{i}$ is greater than the specified tolerance threshold $\theta$ (typically near 100%), and the $\delta = 0$, then the counter for stable iterations of the given input $\boldsymbol{i}$ is incremented (line 12).

If the correlation between $\boldsymbol{o}$ and $\boldsymbol{o}'$ is under the threshold $\theta$, then the number of stable iterations for the given input pattern is set to zero. The threshold $\theta$ can theoretically be enforced to 1.0. This might lead to false instability. The SP internally selects the defined *NumActiveColumnsPerInhArea* of active mini-columns. Two mini-columns with the same overlap will compete for activation (see 5.7.3). The random selection of mini-columns with the same overlap will cause slightly different SDR in subsequent cycles. The HPC algorithm encounters this behaviour and builds in the explicit tolerance defined by $\theta$ less than 1.0. The more efficient approach would be to adopt the SP algorithm to allow side-by-side activation of competing mini-columns. Further exploration and investigation are required for this task in future endeavours.

The second condition for stability is fulfilled when the number of stable iterations, represented by $\gamma_H$, reaches a chosen threshold $\tau$ (line 13) for the entire input space during the training. In the conducted experiments, the threshold $\tau$ was set to 50. However, the actual value chosen varied between 15 and 150 in different experiments. Whenever the correlation value falls below the threshold $\tau$, the counter for stable iterations $\gamma_H$ for that particular input is reset, indicating that the input has not yet achieved the desired stability.

Once the stable state is reached, all generated Sparse Distributed Representations (SDRs) should remain unchanged throughout the lifespan of the Spatial Pooler (SP) instance. The SP is considered stable when two conditions are met: there is a uniform number of active cells in all SDRs, and the required number of stable iterations is reached for all SDRs. The HPC algorithm (Dobric, 2021) continues to monitor stability even after the SP has reached a stable state. The results demonstrate that the extended SP with the HPC algorithm consistently achieves stability with a uniform distribution of active columns for all SDRs.

In Figure 58, the SDRs of two chosen input samples are depicted. The inputs '0' and '1' are both encoded with the stable SDR after approximately 300 iterations. It is observed that the generated SDRs exhibit instability within the first 300 iterations. During this interval, defined as the HTM newborn stage, denoted by the parameter m (line 15), the active columns encoding the SDRs undergo continuous changes. In this stage, mini-column stimulation is active, and SDRs of all inputs frequently change during the learning process (approximately the first 300 hundred cycles in Figure 58.

Following approximately 300 cycles, the HPC mechanism deactivates stimulation, resulting in the rapid convergence of SDRs to a stable state that persists throughout the lifespan of the Spatial Pooler. This experiment involved conducting tests with up to 30,000 iterations, during which the Spatial Pooler remained stable, except for one exception. In some cases, the SP exhibited brief periods of instability shortly after reaching the stable state.

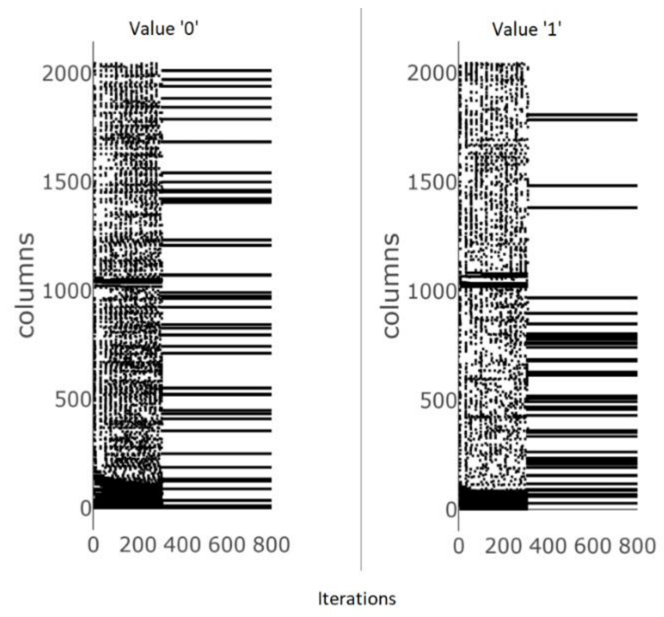


*Figure 58*
*The spatial pooler in the stable state represents two SDRs of two input pattern examples with the activated Homeostatic Plasticity Controller.*

The HPC algorithm is designed to enforce instability when the same processing input undergoes a change in its SDR. In such cases, the HPC algorithm resets the counter of stable iterations for the specific input (line 14), which subsequently declares the Spatial Pooler (SP) as unstable.

In the event of this exceptional occurrence, the learning process can persist until the SP re-enters the stable state for the entirety of its lifespan. This undesired behaviour is predominantly observed when the designated minimum required iterations, denoted as ***m***, are set too low. Opting for larger values of ***m*** appears to resolve this exceptional behaviour; however, it prolongs the duration to transition from the *newborn* stage to the stable state. Nevertheless, even with higher values of ***m***, there is a possibility of slight changes in the Sparse Distributed Representation (SDR) of a pattern over time.

In the HPC algorithm, the stability or instability of the Spatial Pooler (SP) is determined by the tolerance threshold θ. The SP selects the set of active mini-columns based on their overlap, which is calculated by sorting them. During the learning process, certain synapses between columns and input neurons may increase their permanence, leading to an increase in the overlap of specific mini-columns. As a result, these mini-columns are included in the set of active mini-columns. However, due to the required fixed number of mini-columns per SDR, it is possible that previously active mini-columns may be excluded from the set of active mini-columns.
For example, assume four mini-column encode four SDR(i,t) of the input i at the stable iteration t: C1-10, C5-10, C15-9, and C20-9. The first digit denotes the index of the mini-column. The second digit is the overlap of the mini-column in the iteration t. At the current iteration, the mini-column C14 has an overlap of 8. C14-8 is not a part of the SDR because four mini-columns with a higher overlap are chosen. In the next iteration, mini-column C14 increases its overlap to 9. The SDR(i, t+1) might become C1-10, C5-10, C14-9 and C15-9. In this example, the previously active mini-column C20 with the same overlap 9 is randomly replaced with C14-9. The same change will happen if named mini-columns do not change overlap over time. SDR of four mini-columns will be chosen from C1-10, C5-10, C14-9, C15-9, and C20-9. Random choosing of different mini-columns can appear as unstable behaviour to HPC.

By setting the tolerance threshold to θ=1.0, any change in the set of active mini-columns would cause the HPC to transition the stable Spatial Pooler (SP) into an unstable state in the next iteration (t+1). Conversely, with a tolerance threshold of θ=0.75, the SP would remain stable even if a single mini-column within a set of four mini-columns is replaced. Through extensive testing, we discovered that setting θ=0.975, with 40 active mini-columns out of 2048, generates a stable SP. However, higher values of θ may result in temporary instability of the SP after reaching a stable state. Therefore, it is crucial for application developers to carefully select an appropriate value for θ. If the value is not optimally chosen, the HPC algorithm will notify the application when the SP becomes unstable, allowing the application to take necessary actions. In future work, improvements will be made to the HPC algorithm to automatically handle this behaviour without requiring manual intervention.

Figure 59 illustrates this behaviour. In this experiment, the HPC was configured with a relatively low value of m=30 for the minimum required number of iterations during the *newborn* stage. This value is typically considered a very short interval for the *newborn* stage. In the first experiment (Figure 59, left), a threshold of θ=1.0 was utilized, indicating that no changes in the Sparse Distributed Representation (SDR) were allowed to maintain the stable state. The SP entered the stable state at iteration 129 (indicated by the left green line), but it temporarily became unstable at iteration 391 (shown by the left red line) due to replacing a single mini-column.
In the second experiment (Figure 59, right), a threshold of θ=0.975 was used. With this value, it was permissible to replace a single mini-column within the set of 40 chosen active mini-columns. The SP entered the stable state at iteration 84 (indicated by the right green line) in this experiment. Since θ=0.975 allowed a few column replacements during the learning process, the stability of the SP was maintained.

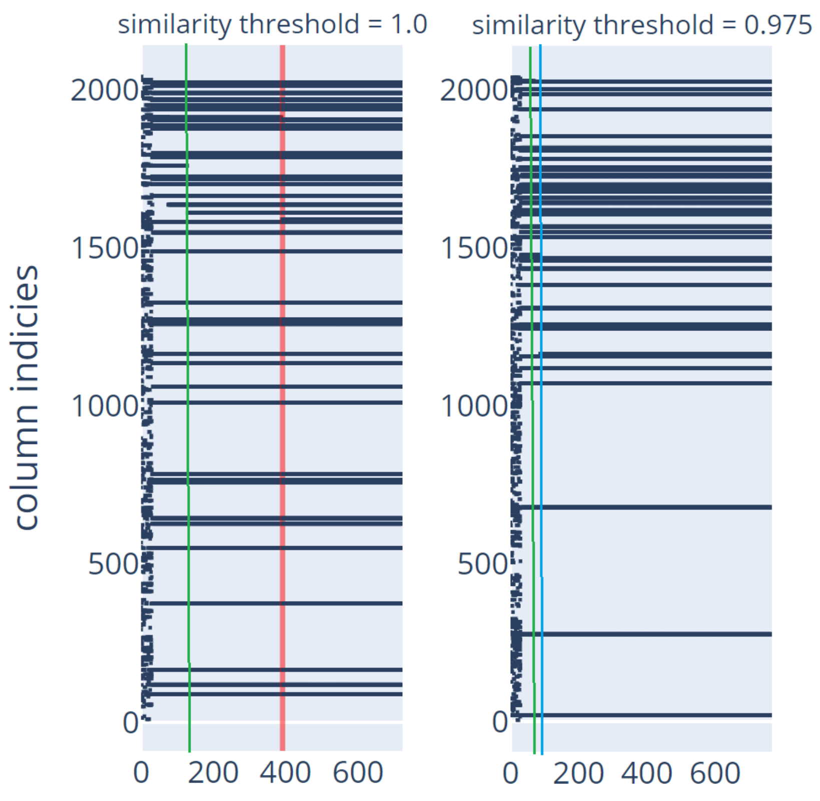


*Figure 59*

*Shortly after entering the stable state, the Spatial Pooler (SP) may experience temporary instability for certain input patterns during a few learning iterations. However, after a few iterations, the SP regains stability and remains in a stable state. The similarity thresholds used in this context are $\theta = 0.97$ and $\theta = 1.0$.*

In Figure 59 (right), the stable state of the Spatial Pooler (SP) is demonstrated, even when some mini-columns are replaced after entering the stable state (indicated by the right blue line). In both cases, the SP exhibits the same behaviour, but the HPC algorithm utilizes a different threshold to determine the iteration step of stability.

This work also examined how stability is achieved throughout the entire input data set. To test this, the number of stable cycles was recorded after a significant period following the start of a stable state. The resulting data trace shown in Figure 60 shows the count of stable cycles for all patterns.

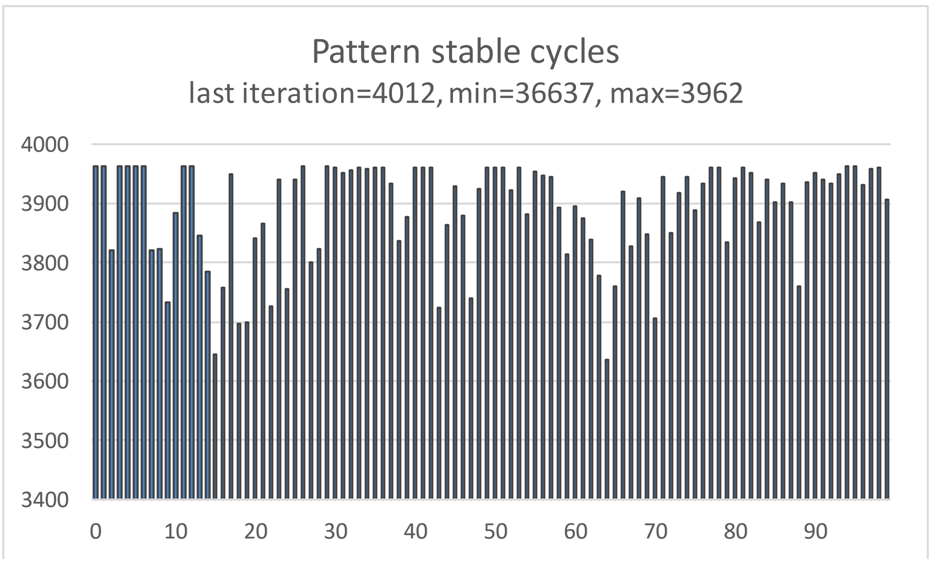


*Figure 60*
*The internal trace of the Homeostatic Plasticity Controller displays the count of stable cycles for each generated Sparse Distributed Representation (SDR) within the entire input data set.*

In this particular experiment, the SP reached the stable state at iteration 441, as shown in Figure 59. The experiment was concluded at iteration 4012, during which the SP remained stable. For the input data set, the minimum number of stable states, 3637, was observed for input 64, while the maximum number, 3962, was observed for multiple numbers such as 0, 1, 2, 3, 4, 5, 6, and more. It should be noted that the SP's stability in these experiments was determined based on a threshold of τ=50 stable iterations. Notably, the maximum number of stable cycles can be calculated by subtracting τ from the total number of iterations, i.e., 4012 - τ. This implies that inputs with the maximum number of stable cycles entered a stable state early in the learning process. This outcome demonstrates that certain patterns achieve stability relatively quickly, while others require more learning cycles to stabilize.

Figure 61 depicts all the SDRs stored in SP after reaching the stable state. In this experiment, the SP was trained using a set of encoded scalar input values ranging from 1 to 100. The x-axis represents the input values from 1 to 100, while the y-axis represents the active mini-columns for each respective input. For instance, the black dots aligned with the green line indicate the SDR code for the scalar input value 60. When input 60 is presented to the SP, the mini-columns along the green line become activated.

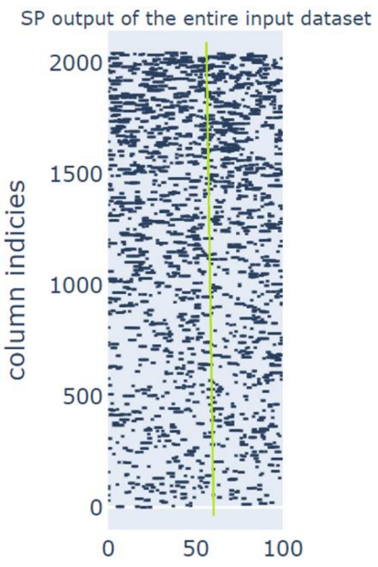


*Figure 61*

*All SDRs at once. Representation of all generated SDRs seen in the training process of 100 input patterns. The horizontal axis shows the index of the input. The vertical axis shows the SDR. Every black "dot" represents the active mini-column.*

This figure shows that the SP uniformly uses the entire mini-column space, resulting from the well-designed plasticity algorithm. As discussed in this chapter, mini-columns wouldn’t be uniformly used without the newborn stage, and the set of active mini-columns across all input patterns would be partially used.

# 9 Discussions

## 9.1 On Parallel Cortical Learning Algorithm with Actor Model

Results in Chapter 6 show that the cortical algorithm can efficiently be scaled with the Actor Programming Model by using higher-level programming languages on commodity hardware. Proposed algorithms SP-MT and SP-Parallel can successfully run on multicore and multi-node architecture, respectively. SP-MT executes on a single-node multicore architecture without the Actor Programming Model and any infrastructure requirement. In contrast, the SP-Parallel algorithm successfully extends the modified version of the CLA and enables it to run in parallel in the cluster. In addition, the specifically modified version of the SP can run internal partial computations for a large number of mini-columns in the Actor Model cluster. Building this algorithm natively in hardware using lower-level programming languages can show better performance when it comes to CPU and RAM usage. However, using widely industrial accepted and efficient higher-level programming languages enable easier use of the computing power of modern cloud environments and enables this technology to a wide community of developers.

Finally, this work shipped the improved version of Spatial Pooler by enabling it for an easy horizontal scale on multiple nodes with support for Windows, Linux, and macOS on almost any kind of hardware. In addition, the Neural Association Algorithm and Temporal Memory can be redesigned and enabled for parallel execution by following the same approach.

## 9.2 On Noise Robustness and Similarity

As previously described, the SP can build the similarity between patterns and achieve noise resistance. Both capabilities work side-by-side and can be controlled by various parameters. Following the results in Chapter 7, the number of non-zero bits in the input vector that encodes a specific spatial pattern greatly influences the noise robustness. Therefore, to better remember some spatial patterns and achieve better robustness on the noise, a higher number of non-zero bits should be used when encoding an input. Biologically, it means that more energy in sensory input leads to higher robustness. However, the increase in the number of non-zero bits will decrease the overall capacity of the memory and a higher probability of false positives. According to equation (5), this trade-off must be considered when building applications.

This finding has a direct impact on the design and implementation of encoders. For example, encoders like scalar encoders, category encoders, etc., should boost encoded input values. This technique uses more active neurons and enables better memorizing and higher resistance against noise (see Figure 50).
However, another technique must be used in the case of images or similar spatial patterns. Therefore, it is proposed to implement a new spatial boosting input layer or a boosting encoder to achieve better results for such kinds of inputs. Note that this suggestion is unrelated to the existing mini-column boosting algorithm described in section 5.7.4.1, which has a different purpose. Furthermore, the proposed spatial boosting input layer should receive encoded input from the encoder appropriately (by keeping the same semantical meaning of input) and increase the number of non-zeros by bolding (striking out) the encoded input and then passing it to Spatial Pooler.

Finally, this component should also ensure that all semantically different patterns have a similar sparseness because different sparseness of inputs will lead to different robustness against noise.

## 9.3 On plasticity and the newborn stage

As previously discussed, the original version of the Spatial Pooler already incorporated a form of homeostatic plasticity mechanism based on previous neuroscience research. However, this initial algorithm exhibited instability during the learning process, posing significant challenges in building reliable applications. Chapter 8 of this work delves into the analysis of this issue and presents a solution by enhancing the existing Spatial Pooler algorithm with a novel component called the Homeostatic Plasticity Controller. The development of the enhanced Spatial Pooler was inspired by observations from neuroscience research, which documented the role of homeostatic plasticity mechanisms during species' developmental stages. Taking cues from these findings, the Homeostatic Plasticity Controller introduces the concept of a "*newborn* stage" in the Spatial Pooler. During this stage, the algorithm emulates the behaviour observed in many species during their early developmental phases. This research underscores the significance of heightened plasticity activity during the initial stages of cortical tissue development. However, this plasticity mechanism brings about a challenge known as the "stability-plasticity dilemma," where the stability of the learning process can be disrupted. Nonetheless, the plasticity mechanism ensures that all mini-columns in the experimental tissue are activated in a balanced manner.

As experiment results show, many mini-columns would not participate in the learning process without plasticity. In that case, parts of the cortical tissue would not be used, and encoding to SDR would not be efficient.

Therefore, during the *newborn* stage, the algorithm stimulates the inactive mini-columns and synapses connected to input neurons. The HPC algorithm injected in the SP waits for a specified number of iterations (*newborn* stage) and then switches off the synaptic and mini-column stimulation mechanism waiting on the SP to enter the stable state. This behaviour corresponds to a growth of the cortical area to the mature form.

By adopting this approach, the Spatial Pooler achieves rapid convergence to a stable state. Applications can subscribe to an event that notifies them about changes in the SP's state. Although the immediate disabling of boosting is not a natural mechanism, it can be easily adjusted if necessary to facilitate a more gradual transition out of the newborn stage, mimicking a biological process.

In conclusion, the original version of the Spatial Pooler (SP) exhibited an unexpected instability resembling an "epileptic" condition. The new Spatial Pooler, incorporating the Homeostatic Plasticity Controller (HPC) and the concept of a *newborn* stage controlled by the HPC, significantly enhances the learning capabilities of the SP and enables the development of more reliable solutions.

## 9.4 On Cortical Learning Algorithm

Previous chapters described various capabilities like SDR encoding, Contextual Associations, Spatial Pooling, and Temporal Memory. In addition, it was shown how different algorithms inspired by biological findings (see Chapter 2) can form artificial tissue called area and how HTM (see Chapter 3.1) integrates into the cortical algorithm.

The *Neural Association Algorithm* introduced and described in section 5.8 is designed as a theoretical framework for learning associations between populations of cells. It is capable of storing a massive number of spatial patterns, can learn sequences and create contextual associations between SDRs.

In experiments in this work, the algorithm focuses mainly on associations with the *Spatial Context* and the *Temporal Context*. This context selection is because the SP and TM are completely implemented parts of this cortical algorithm within *neocortexapi*. The *Spatial Context* creates associations, and the *Temporal Context* predicts the next state. The prediction is modelled as a depolarization of cells that corresponds to biological findings related to basal segments described in section 2.5.

However, the NAA does not propose a strict kind of SDR used as input. This work uses SDR-s encoded by SP or TM. However, the NAA might also use some other type of SDR as the input. For example, some work  (Hawkins, Lewis, Klukas, Purdy, Ahmad, 2019) proposes layer six as a cortical grid cell location. That means that in L6, a different encoding mechanism might be used than the SP encoding in L4. This assumption claims that the location encoded by grid cells would create the *Location Context* in the NAA. Using grid cells in L6 is currently not implemented, but it is a part of future work.

This idea leads to the hypothesis that different encodings in different layers or areas establish associations in different contexts. Still, their encoded SDRs are involved in learning the same way, independent of their origin. This algorithm design enables the same cortical algorithm to handle different problems, unlike traditional ML algorithms discussed in Chapter 3.

Two populations of firing neurons without associations in NAA exist without meaning. Established association between populations of active neurons acts in both directions as a context. For example, population 2 gives some meaning (context) to associated population 1. Associating firing populations set the context and provides a contextual meaning between populations. Giving the meaning to the population is very similar to the labelling task in the supervised algorithms mentioned in Chapter 3. This is an exciting finding because the unsupervised cortical algorithm NAA creates the context which characterizes the supervised learning in every association step. Although the algorithm works unsupervised, it implicitly establishes supervised learning.

Further, associations across multiple regions can create higher-level contextual states that build more complex semantics and meaning. That is when the encoded contextual information from other regions is associated with apical dendrites described in section 2.5.

The following sections discuss how the NAA  can be used to model the canonical unit, to universally solve higher-level tasks and different problems addressed in chapters 2 and 3. First, section 9.4.1 discusses how multiple hierarchical levels can be used to recognize the sequence and the behaviour according to discussed biological findings. Second, section  9.4.2 discusses how associations in NAA are used to build contextual semantics.

### 9.4.1 Complex cells as a temporal association

As described in section 5.9, the Temporal Memory algorithm is a part of the *Neural Association Algorithm*. By design, any encoding that produces SDR can be used as an input for learning sequences. Learning a sequence introduces the temporal dependency, which predicts the next state. That means the series of elements (events) can be contextually associated with time. For example, the sequence of elements "ABCDEFG" can represent just an abstract sequence, a Peptides Sequence that encodes the cancer information, or similar. Even videos are a sequence of encoded frames. Section 5.9 describes sequence learning in TM and how elements create the Temporal Path. The Temporal Path semantically connects sequence elements in a single unit in the order as they appear. For example, frames of a video are temporally chained, indicating that they belong to the same video. If the TM learned many videos, selecting a Temporal Path will look up a single video.

The question of interest in this context is, is it possible to achieve a single SDR representing the Temporal Path of the sequence or some part of it? Section 2.6 discusses two main encoding theories, a localist and a dense theory. The localist theory assumes that a mental state is ideally encoded by a single neuron called the grandmother neuron (Janifer Aniston neuron). In contrast, a dense theory assumes that a single neuron encodes multiple mental states. The following example shows how the whole sequence can be encoded as a single SDR using sequence learning with multiple levels without favouring any named views.
Furthermore, according to Hubel and Wiesel (see section 2.6), lower levels change states frequently when the sequence is recognized, while the topmost layer keeps stable, indicating the identified sequence. The following example explains how NAA naturally performs this task. According to Hubel and Wiesel, sequence learning with temporal associations takes the role of simple cells (lower layers) and complex cells

(higher layers). Figure 62 illustrates how this can be achieved with NAA and Temporal Memory. This unique SDR corresponds neither to localists nor to the dense theory. Moreover, it uses a set of neurons that might be used for encoding multiple states (sequences), but it also forms a unique SDR that encodes a unique sequence (mental state).

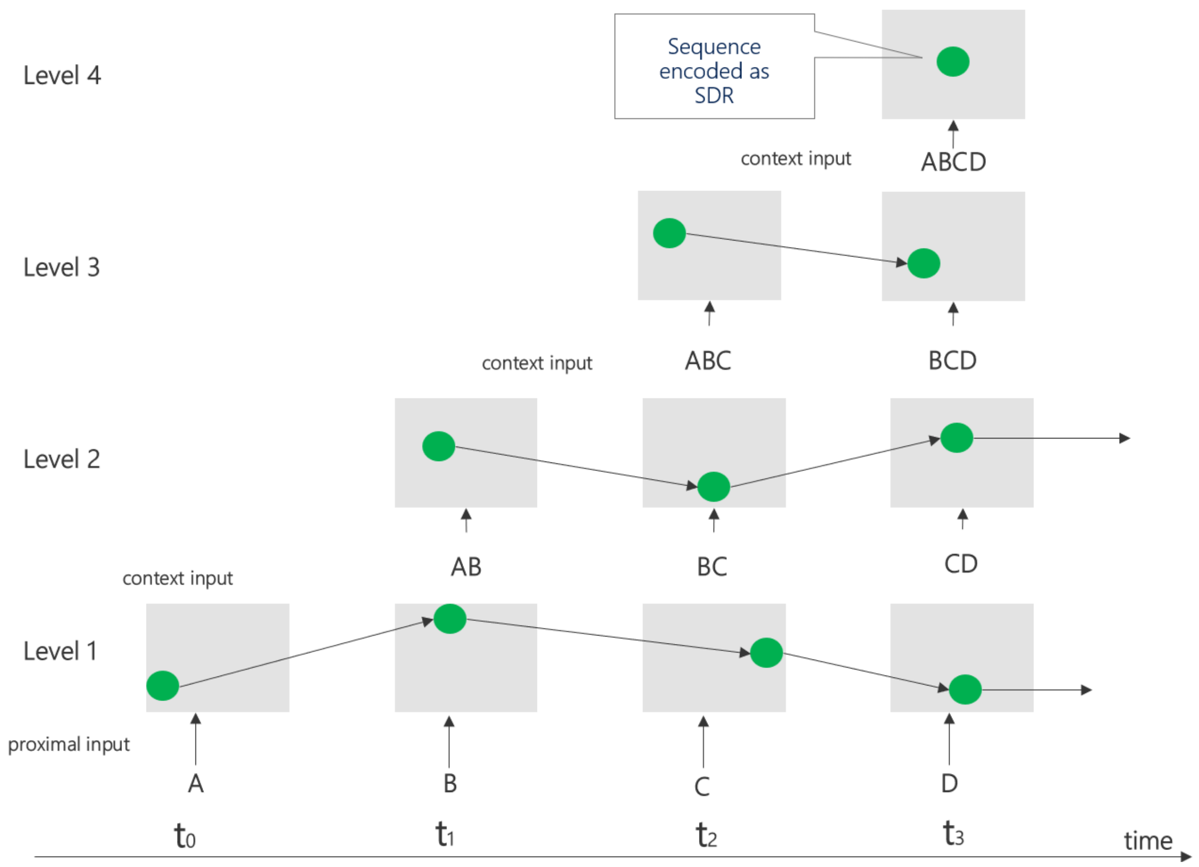


*Figure 62*
*Learning sequence at multiple levels. The last level recognizes the whole sequence by activating a single SDR.*

The sequence is learned at Level 1, as described in section 5.9. Every learned state (SDR) as the output of Level 1 represents the transition semantically from one element to some other element. For example, the green dot at Level 1 (set of multiple active cells) in the cycle at time t1 represents the transition from element A to element B. This state encoded as a set of active cells can be used as an input for the next Level 2. Level 2 starts with the element, which is SDR produced at Level 1. SDR made at Level 1 means the transition from A to B (noted as "AB" at Level 2 in cycle t1). In the next cycle, t2, Level 2 will learn the SDR “BC” after the SDR “AB”. That means Level2 will form at the cycle t2, the SDR, representing the subsequence “ABC”. This rule can be applied to all elements at multiple levels. Finally, at Level 4, the produced SDR at the cycle t3 will encode the whole Sequence “ABCD”, whereas lower levels will encode parts of the sequence.

This example shows that NAA-modelled sequence learning in this work can recognize more complex sequences (behaviours) at higher levels. Theoretically, connecting cortical areas to endless hierarchy levels would be able to remember the sequence with infinite elements. In other words, the infinite number of elements could encode the entire universe persisted as a sequence if the same would be repeated a few times to enable learning. Furthermore, such a theoretical brain would be able to recognize all states learned from the beginning as a single sequence. This exciting finding will be a topic of future research.

### 9.4.2 Contextual Associations and a Meaning

Chapter 5 described how the Cortical Learning Algorithm was modelled in this work based on biological findings and ideas of the HTM. It briefly described all artefacts like SDR, Plasticity, Inhibition, Spatial Pooler, Temporal Memory, the Neural Association Algorithm etc. In this work, all crucial components have been implemented and used for experiments presented in this document. The general Neural Association Algorithm that connects multiple areas is still a work in progress, but it already introduces exciting ideas that might help to understand how meaning is created in the neocortex and the brain.

The algorithms described in this work are all designed strictly aligned with biological findings. Patterns are continuously learned (SP) and temporally associated (TM).

As mentioned, two patterns encoded as two independent SDRs have no meaning. However, if synaptic connections are formed between two SDRs (see section 5.8), an association is created. In that case, the associating SDR establishes a context and meaning for the associated SDR. That means that two independent SDRs (states) are meaningless until they are associated.

Also, if the SDRs (states) are associated with the same SDR*k*, then the associated SDR*k* formally tags the set of associating SDRs. This kind of association defines supervised learning, which spontaneously happens inside an unsupervised cortical algorithm when associations are created between meaningless SDRs. The SDR$_k$ defines, in this case, some meaning at the higher level in the given context.

For example, if the set of associating SDRs represents images of fruits, then the SDR$_k$ might represent the tag "fruit". The same example can also be hierarchically constructed using multiple levels of learning (Multilevel Learning). This is illustrated in Figure 63. Assume the experiment contains three areas (Cortical Units): Areal Level 1, Area Level 2 and Area Level 3. The Areal Level 1 receives a sensory input that algorithm A1 learns. As discussed, the Neural Association Algorithm does not strictly propose what algorithm must be used to encode the SDR. In this example, the Spatial Pooler encodes the input pattern into a set of active mini-columns represented as SDR. Area Level 1 learns fruits f$_1$ to f$_M$. Every fruit is represented as a set of images $f_{ij}$. At the same time, when presenting the fruit image, the supervised context input is created for Area Level 2 using some algorithm A2. This algorithm encodes the supervised contextual input into the SDR f$_1$ for all images (inputs) f$_1$- f$_N$. The Neural Association Algorithm immediately creates associations from SDRs $f_{ij}$ to a set of active cells f$_1$. It associates populations to a single population and mathematically reduces dimensions simultaneously.

Following the same principle, a set of images of fruit f1 is associated with the single SDR *f1, a* set of images of fruit f2 is associated with the single SDR *f2* etc. Further, SDRs, which represent some specific fruit *f1, f2,.., f1,.., fK* can be associated with another but same SDR *f* that represents a higher level state called, let's say "*all known fruits*". This principle can be further followed. For example, the SDR f that represents "*all known fruits*" can be associated with other higher or even lower-level SDR that represents anything else. This is how meaning is grown by creating a mesh of associations between sensory inputs and contextual SDRs at multiple levels.

Encoding two contextual values together as a paired value does not build meaning. For example, the month can be encoded with the date as a paired value, JAN-15 or FEB-15.
In some industrial scenarios, this is an efficient and helpful technique often used in HTM solutions. It sets the "meaning" to the pair of values. However, such encoding leads to the loss of the meaning of encoded values in their context. For example, if both values were encoded with the scalar encoder (see 5.5), the produced SDR would

spatially build the similarity at the same date (15th in the month), but this does not create an explicit meaningful association to date and month.

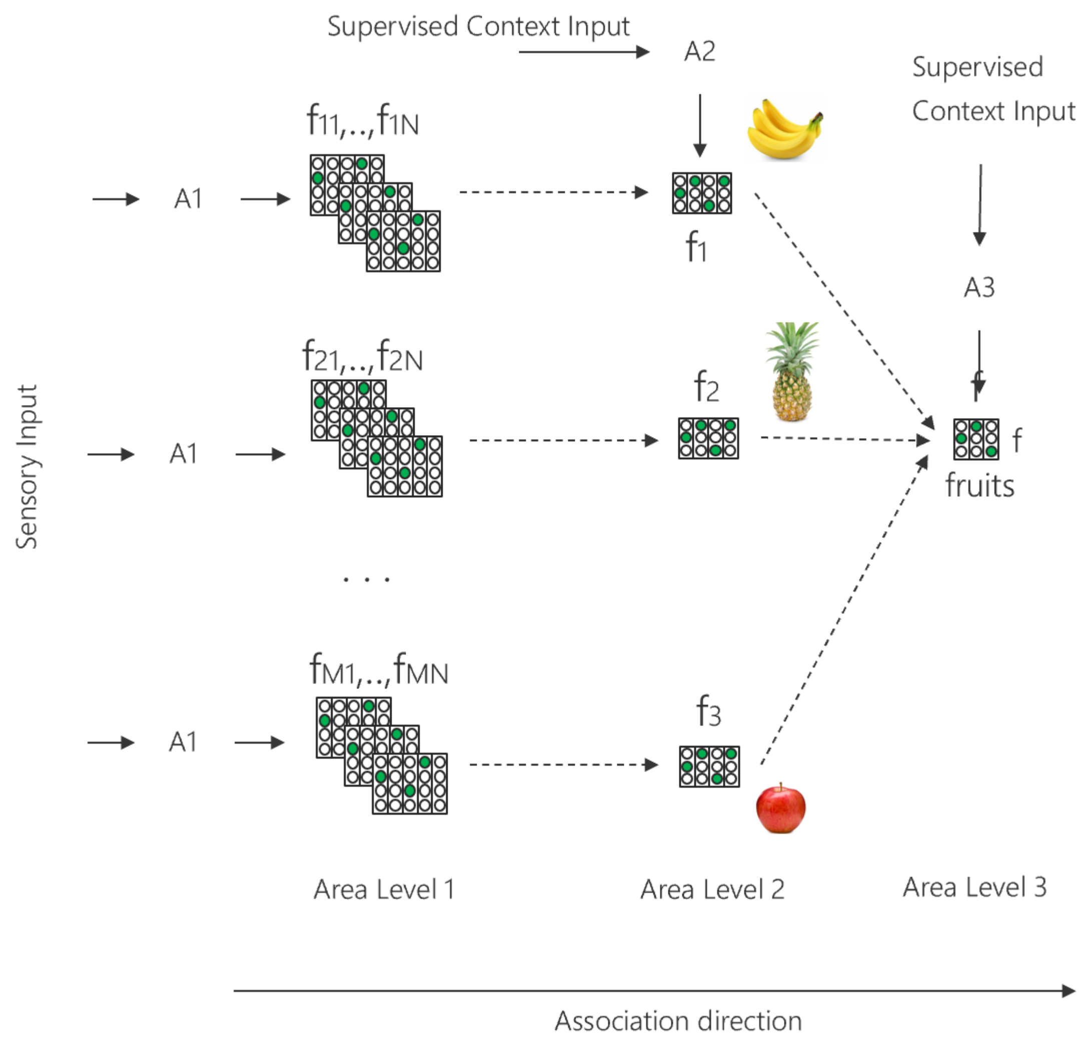


*Figure 63*

*Building semantic associations and meaning by using multiple levels of learning (Multilevel Learning)*

## 9.5 Future Work

Results presented in this work demonstrate that the current version of the NAA is a universal algorithm able to approach solutions for different kinds of problems discussed in chapters 2 and 3. The algorithm described in this work is a theoretical approach that generalizes the HTM. It explains how to model an artificial cortical area as a canonical cortical unit, which can be synoptically connected to a network of units, capable of solving higher-level tasks by establishing contextual associations. At the moment of writing this document, one part of NAA was already implemented in the *neocortexapi* (Dobric, 2019) with full support for Spatial Pooler, Temporal Memory, inhibition, structural plasticity and various encoders. However, the current implementation is limited to a single area only. With this limitation, the algorithm cannot learn any contextual associations but the temporal ones. One of the important requirements for the next version of the *neocortexapi* framework is a complete implementation of the algorithm, as described in section 5.8. This implementation will enable the final validation of the algorithm and implementation of more sophisticated scientific and industrial scenarios. The following sections describe in more detail the tasks that should be targeted in the near future. The future studies are grouped in two areas: Tasks related to core algorithm research and task related to industrial scenarios.

### 9.5.1 Core Algorithm Research

The following sections describe recommended future work related to the further research of core features of the cortical algorithm described in Chapter 5.

#### 9.5.1.1 Finalization of implementation of NAA

To validate the NAA, a complete implementation of the algorithm is required. As mentioned, the current implementation supports the SP, TM, encoders and related inhibition and plasticity algorithms. However, due to the limitation of the single cortical area, the current version can not learn associations that are not temporal (used by TM). Therefore, the framework should be extended to enable a network of cortical areas to make this possible. As discussed in 9.4.2, such a network can build multiple contextual associations and enable the learning of higher-level cognitive tasks. Extending the framework mainly includes support for interconnecting areas by implementing apical synaptic connections (see 2.4). The current implementation does support proximal dendrite segments to sensory input (used by SP) and the distal (basal) dendrite segments (used by TM). The extended version should include support for apical dendrite segments (see 2.5) that will connect cells and mini-columns across different areas, enabling contextual associations and the establishment of meaning between SDRs (see 9.4.2).

#### 9.5.1.2 Reverse Encoding

The typical experiment in this area requires implementing the inferring (prediction) code. This primarily requires reverse encoding from the SDR back to the input value. For example, the set of active mini-columns encoded by the SP or the set of active cells encoded by the TM should be converted back to the spatial input or state transition, respectively. Currently, the *neocortexapi* enables such scenarios using the HtmClassifier component, which is not implemented by aligning with any biological finding. The drawback of this component is the increasing memory by increasing the number of input elements. The full implementation of NAA could use associations and the supervised labelling mechanism described in the previous section to implement the reverse encoding. For example, reverse encoding to the spatial input could be done by establishing associations between the encoder output and the encoded SDR.

#### 9.5.1.3 Feed Forward Network for hierarchical recognizing of sequences

As discussed in 9.4.1, NAA's multiple hierarchically connected feed-forward layers can recognize sequences. This can be achieved by using apical connections (see 9.5.1.1), which will create a Feed-Forward layered cortical structure shown in Figure 62. The output of every layer is the SDR, a set of active cells calculated in this case by the TM. Because every SDR semantically represents a transition from the state, it will be used as a higher-level contextual input (not just an element in the sequence) for the next layer. In this context should also be investigated if using the feedback connection can improve the learning process. This implementation should validate that NAA can produce results originally found by (Hubel, Wiesel, 1959).

#### 9.5.1.4 Invariant Object Representation

As discussed in chapters 5.7 and 7, Spatial Pooler recognizes spatial patterns, which group patterns by spatial similarity, but does not provide any semantic relation between them. To semantically associate patterns in a specific context, the SP SDR should be connected with other contextual areas of different sizes of the receptive field.

For example, SP1 can learn a specific pattern in the receptive field (see 2.3) in size 28x28, and SP2 can learn the same pattern in size 64x64. The same principle can be applied to rotated patterns etc. All SPs that learn a different pattern variation will build associations to the same pattern that represents a context in a different area, as discussed in section 9.4.2. The set of all learned SPs can be used to predict spatial patterns in a learned position.

#### 9.5.1.5 Implementation of Grid-Cells

The current version of the algorithm and its framework implementation uses various encoders described in section 5.5. The information observed by theoretically any sensor can be encoded to SDR, which is used as input for further processing. The encoded input is typically used by the Spatial Pooler to create the SDR representation in the form of active mini-columns, which TM uses to produce the cell SDR. However, the NAA does not explicitly propose how the SDR of active cells should be created. That means any kind of encoding can be used inside an area to represent some mental or other state. For example, one of the encoding mechanisms can be implemented with the help of Grid Cells (see 9.4), which typically encode a location but can be used to encode mental states. This task aims to provide support for Grid Cell encoding inside of NAA and investigate the impact on learning of associations.

#### 9.5.1.6 Applying Alpha and Gamma Cycles and error feedback

There is biological evidence that the learning process in the brain is organized in phases within alpha and gamma cycles (O'Reilly, Wyatte, Rohrlich, 2017). Alpha cycles are defined by 100ms (10Hz) oscillations and typically characterize the dynamics of the deep neural network. In contrast, superficial network layers are characterised by Gamma oscillations of 25ms (40Hz). The current version of NAA does not make explicit usage of both cycles. However, as discussed in section 9.4.1, learning at lower levels is faster because every next higher level must wait on the result of the previous level.

The framework Deep Leabra discussed in section 3.2 provides a model that uses named cycles. It should be investigated if and how alpha and gamma oscillations occur in NAA, intending to understand their meaning better. Another interesting aspect of this framework that needs to be examined is using the feedback error.

#### 9.5.1.7 Investigating Inhibition

In this research, multiple inhibition algorithms have been tried. However, no significate influence on the performance has been noticed. Therefore, in future work, the inhibition algorithm should be more focused on analysing how the inhibition process influences the sparsity and the overall performance of the learning process.

#### 9.5.1.8 Hardware Implementation

As described in Chapter 6, this work delivers a proposal and the framework for running the algorithm in parallel clusters of many nodes. However, to increase performance, implementing the NAA on the hardware platform could enable the implementation of new scenarios.

### 9.5.2 Industrial Scenarios

This section describes some important and valuable industrial scenarios that should/can be implemented using NAA and HTM.

#### 9.5.2.1 Anomaly Detection

The capability of learning sequences by creating the temporal contextual association (see 5.9) can be used for anomaly detection scenarios. For example, the sequence of elements ABCDEFG or numbers 1.5, 75, 9, and 3.5 can be learned by NAA. As described in section 5.9, the algorithm holds a set of predictive cells in any learning cycle. When element B appears, the set of predictive cells that encode the next element C will be depolarized etc. The anomaly detection solution can easily be implemented by observing the next predicted element (see *ComputeCycle.PredictiveCells* in *neocortexapi* framework) and comparing it with the actual element. When predicting numbers like power consumption or similar predictions, a more complex solution can be created. In such cases, the NAA can predict precisely the expected learned value, like 123.45. The anomaly detection code should be optimized for the specific industrial scenario and allow some reasonable tolerance. For example, similar to the scalar encoder (see 5.5), all values in the range 120.00-125.00 should not be detected as an anomaly. With this approach, values are not encountered as discrete values but rather with a configurable overlap between nearby values (see Figure 22).

#### 9.5.2.2 Video Detection and Image Classification

The capability of the sequence learning and neural associations makes possible implementation of the video learning algorithm easy. The NAA algorithm implementation in the *neocortexapi* already contains a component for encoding the image into the SDR that can be used as input of the Spatial Pooler. The incoming video stream should be split into a sequence of frames, encoded by the image encoder to SDR and learned by NAA. The learning can be done by sequence learning or by associative learning. Every learned frame should be associated with the video. The prerequisite for this experiment is the full implementation of NAA, as described in section 9.5.1.1.

## 9.6 Summary of novelties and important findings

This section summarizes all the important findings and novelties of this work.

1. Understanding and applying the biological encoding of sensory input to SDR (see 5.5).

2. Proposal for Neural Association Algorithm as a generalization for HTM-CLA (see sections 5.8 and 9.4).

3. Understanding and controlling the robustness to noise and similarity in Spatial Pooler (see sections 7 and 9.2).

4. Understanding of the instability of the Spatial Pooler due to structural plasticity (see Chapter 8).

5. Solving the instability of the SP by introducing the *newborn* stage (see Chapter 8).

6. Proposal for solving plasticity-stability dilemma (see section 9.3).

7. Proposal for spatial and temporal representation of the state defined by SDR (see section 5.9, Figure 33).

8. Proposal for definition of meaning and contextual associations (see section 9.4.2).

9. Proposal on how sequence learning and temporal associations can enable higher-level meaning, akin to how Complex Cells function (see section 9.4.1).

10. Design and implementation of the opensource framework neocortexapi in C# with support for Encoders, Spatial Pooler, Temporal Memory, Homeostatic Plasticity Controller (*newborn* stage), HtmClassifier (SDR reverse encoding), Neural Association Algorithm and Predictor for Multisequence Learning (see section 5.8).

11. Design and implementation of the lightweight Actor Model Framework for cortical calculation (see 6.2).

# 10 References

Abraham, Jones, Glanzman, 2019. Is plasticity of synapses the mechanism of long-term memory storage?. *npj - sciense of learning,* Volume 4, p. 9.

Actors, A., 2015. *Akka Actors.* [Online]
Available at: https://doc.akka.io/docs/akka/current/typed/actors.html

Ahmad, Hawkins, Cui, 2017. A Theory of How Columns in the Neocortex Enable Learning the Structure of the World. *Frontiers in Neural Circuits,* Volume 11, p. 81.

Ahmad, Lavin, Purdy, 2017. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing,* Volume 262.

Andrews, G. R., 1998. *A Method for solving synchronization problems.* s.l., s.n., pp. 1-21.

Anon., 2009. *Microsoft Azure Service Bus.* [Online]
Available at: https://azure.microsoft.com/en-us/services/service-bus/

Anon., 2019. *Developer Survey Results.* [Online]
Available at: https://insights.stackoverflow.com/survey/2019

Arcas, Fairhall, Bialek, 2003. Computation in a single neuron: Hodgkin and Huxley revisited. *Neural Comput.,* Volume 15.

Arora, Basu, Mianjy, Mukherjee, 2018. *Understanding Deep Neural Networks with Rectified Linear Units.* Vancouver, ICLR.

Barth, Poulet, 2012. Experimental evidence for sparse firing in the neocortex. *National Library of medicine,* pp. 345-55.

Bell, Sejnowsky, 1997. Edges are Independent Components of natural Scenes. *Sciense Direct,* Volume 37, pp. 3327-3338.

Bennett, M., 2020. An Attempt at a Unified Theory of the Neocortical Microcircuit in Sensory Cortex. *Frontiers in Neural Circuits,* Volume 14.

Bernstein, Bykov, Geller, Kliot, Thelin, 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* [Online]
Available at: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

Bonhof, G. M., 2008. *Using Hierarchical Temporal Memory for Detecting Anomalous Network Activity,* s.l.: AIR FORCE INSTITUTE OF TECHNOLOGY.

Bowers, 2009. On the biological plausibility of grandmother cells: Implications for neural network theories in psychology and neuroscience. *Psychological Review,* Volume 116, pp. 220-251.

Buxhoeveden, Casanova, 2002. The minicolumn and evolution of the brain. *Brain, Behavior and Evolution,* 60(60), pp. 125-151.

Buxhoeveden, Casanova, 2002. The minicolumn hypothesis in neuroscience. *Brain,* 125(5), pp. 935-951.

Chadha, Dobric, 2020. *dotnetactors.* [Online]
Available at: https://github.com/ddobric/dotnetactors
[Accessed 2022].

Chen; Lui., 2021. Neuroanatomy, Neuron Action Potential. *NCBI.*

Chklovski, Mel, Svoboda, 2004. Cortical rewiring and information storage. *Nature,* p. 782–788.

Cui, Ahmad, Hawkins, 2018. The HTM Spatial Pooler—A Neocortical Algorithm for Online Sparse Distributed Coding. *Frontiers in Computational Neurosciences,* 29 11, Volume 11, p. 111.

Damir Dobric, Andreas Pech, Bogdan Ghita, Thomas Wennekers, 2021. *Improved HTM Spatial Pooler with Homeostatic Plasticity Control.* Vienna, SciTePress, pp. 98-106.

Davis, G. W., 2013. Homeostatic Signaling and the Stabilization of Neural Function. *Neuron,* p. 09.044.

Dietmar Heinke, Peter Wachman, Wieske von Zoest, E. Charles Leek, 2021. A failure to learn object shape geometry: Implications for convolutional, neural networks as plausible models of biological vision. *Journal of experimental psychology. Animal learning and cognition,* Volume 189, pp. 81-92.

Dobric, Pech, Ghita, Wennekers, 2020. *On the Relationship Between Input Sparsity and Noise Robustness in Hierarchical Temporal Memory Spatial Pooler.* Rom, ESSE, pp. 187-193.

Dobric, Pech, Ghita, Wennekers, 2020. *Scaling the HTM Spatial Pooler.* s.l.:s.n.

Dobric, Pech, Ghita, Wennekers, 2021. *Improved HTM Spatial Pooler with Homeostatic Plasticity control*. Vienna, s.n.

Dobric, Pech, Ghita, Wennekers, 2022. On the Importance of the Newborn Stage When Learning Patterns. *Springer Nature Computer Sciences,* Volume 3, p. 179.

Dobric, Pech, Wennekers, Ghita, 2020. *Prallel Spatial Pooler with Actor Programming Model*. Helsinki, AIS 2020 - 6th International Conference on Artificial Intelligence and Soft Computing.

Dobric, 2018. *GitHub*. [Online]
Available at: https://github.com/ddobric/neocortexapi

Dobric, 2021. *Implementation of Homeostatic Plasticity Controller*. [Online]
Available at:
https://github.com/ddobric/neocortexapi/blob/master/NeoCortexApi/NeoCortexApi/HomeostaticPlasticityController.cs

Dobric, 2021. *Implementation of Homeostatic Plasticity Controller*. [Online]
Available at:
https://github.com/ddobric/neocortexapi/blob/42e73b282e848a9f8028954377b0a2af3c08570e/source/NeoCortexApi/HomeostaticPlasticityController.cs

Dobric, D., 2016. *Modern Backends Service Fabric & Actor Model*. [Online]
Available at: https://slideplayer.com/slide/13072159/

Dobric, D., 2016. *Modern Backends with Actor Programming Model*. Heidelberg, Heise Verlag.

Dobric, D., 2019. *NeoCortexApi*. [Online]
Available at: https://github.com/ddobric/neocortexapi/blob/master/README.md

Dobric, D., 2019. *NeoCortexApi Entities*. [Online]
Available at:
https://github.com/ddobric/neocortexapi/tree/master/source/NeoCortexEntities

Douglas, Martin, 1989. A Canonical Microcircuit for Neocortex. *Neural Computation,* Volume 4, pp. 480-488.

Douglas, Martin, 2004. Neuronal Circuits of the neocortex. *Annual Review of Neuroscience,* Volume 27, pp. 419-451.

Faisal, Selen, Wolpert, 2008. Noise in the nervous system. *Nature Reviews Neuroscience,* Volume 4, pp. 292-303.

Ferrier, M., 2014. *Toward a Universal Cortical Algorithm: Examining Hierarchical Temporal Memory in Light of Frontal Cortical Function,* s.l.: arXiv.

Finelli, Haney, Bazhenov, Stopfer, Sejnowski, 2007. Synaptic Learning Rules and Sparse Coding in a Model Sensory System. *PLOS Computational Biology.*

Fukushima, K., 1980. Neocognitron: A Self-organizing Neural Network Model. *Biological Cybernetics,* Issue 36, pp. 193-202.

George, Hawkins, 2009. Towards a Mathematical Theory of Cortical Micro-circuits. *PLOS Computational Biology,* 5(10).

Gerstner, Kempter, Hemmen, Wagner, 1996. A neuronal learning rule for sub-millisecond temporal coding. *Nature,* Volume 383, pp. 76-78.

Graeme. Davis, 2006. Homeostatic Control of Neural Activity - From Phenomenology to Molecular Design. *The Annual Review of Neurosciences,* Volume 29, p. 307–23.

Guo, Xiang, Zhang, Su, 2021. Integrated Neuromorphic Photonics: Synapses, Neurons,and Neural Networks. *Advanced Photonic Research,* 2(6).

Guo, Xie, Zhang, Li, 2015. *Disease Diagnosis Supported by Hierarchical Temporal Memory.* s.l., UIC-ATC-ScalCom-CBDCom-IoP.

Guy, Staiger, 2017. The Functioning of a Cortex without Layers. *Frontiers in Neuroanatomy,* Volume 11, p. 54.

Harold, Kirchner, 1974. *Pathology of the Ear.* s.l.:Harvard University Press, Cambridge, Massachusetts.

Hasselmo, M. E., 2013. *How we remember - Brain Mechanisms of Episodic memory.* s.l.:MIT Press.

Hawkins, Ahmad, Dubinsky, 2011. *HTM Cortical Learning Algorithms.* [Online] Available at: https://www.numenta.com/htm-overview/education/HTM_CorticalLearningAlgorithms.pdf

Hawkins, Ahmad, Subutai, Yuwei, 2017. A Theory of How Columns in the Neocortex Enable Learning the Structure of the World. *Frontiers in Neural Circuits,* Volume 11, p. 81.

Hawkins, Ahmad, 2016. Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex. *Frontiers in Neural Circuits,* Volume 10, p. 13.

Hawkins, Blakeslee, 2004. *On Intelligence: How a New Understanding of the Brain Will Lead to the Creation of Truly Intelligent Machines.* 1 ed. s.l.:Macmillan Publishers.

Hawkins, Dawkins, 2021. *A Thousand Brains: A New Theory of Intelligence.* s.l.:A Thousand Brains: A New Theory of Intelligence.

Hawkins, Lewis, Klukas, Purdy, Ahmad, 2019. Framework for Intelligence and Cortical Function BasedA on Grid Cells in the Neocortex. *Frontiers in Neural Circuits,* Volume 12.

Hearst, Karger, Pedersen, 1996. Scatter/Gather as a Tool for the Navigation of Retrieval Results. *Research Gate.*

Hebb, D., 1949. *The organization of behavior; a neuropsychological theory.* s.l.:American Psyhology Assotiation.

Herculano-Houzel, S., 2009. *The Human Brain in Numbers: A Linearly Scaled-up Primate Brain.* [Online]
Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2776484/

Hewitt, Bishop, Steiger, 1973. *A Universal Modular ACTOR Formalism.* s.l., Artifficiial Intelligence.

Hewitt, C., 2010. Actor Model for Discretionary, Adaptive Concurrency. *CoRR,* Volume abs/1008.1459, p. 25.

Hinton, G., 2021. How to represent part-whole hierarchies in a neural network - GLOM. *CoRR.*

Hinton, G. F., 1981. A Parallel Computation that Assigns Canonical Object-Based Frames of Reference. *ACM Digital Library,* Volume 2, pp. 683-685.

Hochreiter, Schmidhuber, 1997. Long Short-Term memory. *Neural computation,* 9(8), pp. 1735-1780.

Hoffmann, H., 2019. Heiko Hoffmann. *Neural Computation,* 31(5).

Hromádka, Weese, Zador, 2008. Sparse Representation of Sounds in the Unanesthetized Auditory Cortex. *PLOS Digital Health.*

Hubel, Wiesel, 1959. Receptive fields of single neurones in the cat's striate cortex. *Journal of Psyhology.*

Hubel, Wiesel, 1962. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex.. *Journal of Physiology,* Volume 160, pp. 106-154.

Hubel, Wiesel, 1968. Receptive fields and functional architecture. *Journal of Psychology,* Volume 195, pp. 215-243.

Hubel, Wiesel, 1974. Uniformity of monkey striate cortex: a parallel relationship between field size, scatter, and magnification factor. *Journal of comparative neurology,* 158(3), pp. 295-305.

Hubel, Wiesel, 1974. Uniformity of monkey striate cortex: A parallel relationship between field size, scatter, and magnification factor. *JCN,* 158(3), pp. 295-305.

Hubel, 1995. *Eye, Brain, and Vision.* s.l.:Cientifical Americn Library.

Jonathan Lejeune, L. A. J. S. P. S., 2015. *Reducing synchronization cost in distributed.* s.l., s.n., pp. 540-549.

Kaas, J. H., 2012. Evolution of columns, modules, and domains in the neocortex of primates. *PNAS,* 109(10655–10660), p. 6.

Kajić, Gosmann, Stewart, Wennekers, Eliasmith, 2017. A Spiking Neuron Model of Word Associations for the Remote Associates Test. *Frontiers in Psychology,* Feb, Volume 8, p. 14.

Larkum, Nevian, 2008. Synaptic clustering by dendritic signalling mechanisms. *Science Direct,* Volume 18, pp. 321-331.

Larkum, Nevian, 2008. Synaptic clustering by dendritic signalling mechanisms. *PubMed,* 18(3), pp. 321-31.

LeCun, Bengio, Hinton, 2015. Deep Learning. *Nature,* Issue 521, pp. 436-44.

LeCun, Y. a. C. C., 2010. *MNIST handwritten digit database}.* [Online]
Available at: http://yann.lecun.com/exdb/mnist

Leijen, Schulte, Burckhardt, 2009. *The Design of a Task Parallel Library,* s.l.: Microsoft Research.

Lewis, Purdy, Ahmad, Hawkins, 2019. Locations in the Neocortex: A Theory of Sensorimotor Object Recognition Using Cortical Grid Cells. *Frontiers in Neural Circuits,* Volume 13, p. 22.

Lücke, Bouecke, 2005. *Dynamics of Cortical Columns – Self-organization of Receptive Fields.* s.l., ICANN.

Lücke, J., 2004. *Hierarchical Self-Organization of Minicolumnar Receptive Fields.* s.l., s.n., pp. 1377-89.

Lyon, J. H. K. a. D. C., 2001. *Visual cortex organization in primates: theories of V3 and adjoining visual areas.* [Online]
Available at: http://invibe.net/biblio_database_dyva/woda/data/att/bbbd.file.pdf

Maffei, Nelson, Turrigiano, 2004. Selective reconfiguration of layer 4 visual cortical circuitry by visual deprivation. *PubMed,* Volume 2.

Marianne Fyhn,Torkel Hafting,Menno P. Witter,Edvard I. Moser,May-Britt Moser, 2008. Torkel Hafting, Marianne Fyhn, Sturla Molden, May-Britt Moser & Edvard I. Moser. *Hippocampus,* 18(12), pp. 1230-1238.

Martial, Aurélia, Patrick, 2013. The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects. *Frontiers in Psychology,* Volume 4.

McCulloch, Pitts, 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics,* Volume 5, pp. 115-133.

Melis, Chizuwa, Kameyama, 2009. *Evaluation of Hierarchical Temporal Memory for a RealWorldApplication.* s.l., s.n.

Microsoft Corporation, 2016. *Service Fabric Reliable Actors.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-introduction

Microsoft, 2019. *Orleans is a cross-platform framework for building robust, scalable distributed applications.* [Online]
Available at: https://dotnet.github.io/orleans/

Microsoft, 2020. *Mutex class.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=net-6.0

Microsoft, C., 2021. *Durable Entities.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp

Mike, B., 2006. *The Chubby lock service for loosely-coupled distributed systems.* s.l., Google.

Mishra Tania Banerjee, Sahni Sarta, 2013. *An efficient publish/subscribe system.* s.l., IEEE Symposium on Computers and Communications (ISCC).

Mitchell, B., 2005. *Resolving race conditions in asynchronous partial order scenarios.* s.l., IEEE Transactions on Software Engineering, pp. 767-784.

Mnatzaganian, Fokoué, Kudithipudi, 2016. *A Mathematical Formalization of Hierarchical Temporal Memory's Spatial Pooler.* [Online]
Available at: https://arxiv.org/abs/1601.06116

Moon, Chang, 2006. A thread monitoring system for multithreaded Java programs. *ACM SIGPLAN Notices,* pp. 21-29.

Mountcastle, V., 1957. Modality and topographic properties of single neurons of cat's somatic sensory cortex.. *Neurophysiol,* Volume 20, pp. 408-34.

Mountcastle, V. B., 1997. *The columnar organization of the neocortex.* [Online]
Available at: http://invibe.net/biblio_database_dyva/woda/data/att/340e.file.pdf

Numenta, 2008. *NUPIC.* [Online]
Available at: https://github.com/numenta/nupic

O'Reilly, Wyatte, Rohrlich, 2017. *Streams, Deep Predictive Learning: A Comprehensive Model of Three Visual.* [Online]
Available at: Randall C. O'Reilly and Dean R. Wyatte and John Rohrlich

Olshausen, Field, 1996. Emergence of simple cell receptive field. *Letters to Nature,* Volume 381.

open-source, htm-java, 2013. *HTM.JAVA.* [Online]
Available at: https://github.com/numenta/htm.java

Pal, Bhattacharya, Dey, Mukherjee, 2018. *Modelling HTM Learning and Prediction for Robotic Path-Learning.* s.l., 7th IEEE International Conference on Biomedical, pp. 26-29.

Pelluru, Schneider, Wolf, 2021. *Message Sessions.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/service-bus-messaging/message-sessions

Perrinet, Laurent, 2010. Role of homeostasis in learning sparse representations. *Neural Computation,* Volume 22, p. 36.

Petabridge, 2016. *Akka.NET*. [Online]
Available at: https://getakka.net/

Pethick, Liddle, Werstein, Huang, 2003. *Parallelization of a Backpropagation Neural Network on a Cluster Computer*. [Online]
Available at:
https://www.researchgate.net/publication/228549385_Parallelization_of_a_Backpropagation_Neural_Network_on_a_Cluster_Computer

Pietron, Wielgosz, Wiatr, 2016. *Parallel Implementation of Spatial Pooler in Hierarchical Temporal Memory*. s.l., s.n., pp. 346-353.

Quiroga, Kreiman, 2010. Measuring sparseness in the brain. Comments on Bowers (2009). *NCBI*.

Quiroga, R. Q., 2017. *The Forgetting Machine: Memory, Perception, and the "Jennifer Aniston Neuron"*. s.l.:BenBella Books.

Rhen, Sommer, 2006. A network that uses few active neurones to code visual input. *Springer,* pp. 135-56.

Rodriguez, Whitson, Granger, 2004. Derivation and Analysis of Basic Computational Operations of Thalamocortical Circuits. *Journal of cognitive neurosciences,* 16(5), pp. 856-877.

Sachs Kai and Appel, Stefan and Kounev, Samuel and Buchmann, Alejandro, 2010. *Benchmarking Publish/Subscribe-Based Messaging Systems*. s.l.:s.n.

Sebastian Billaudelle, Subutai Ahmad, 2016. *Porting HTM Models to the Heidelberg Neuromorphic*. s.l.:s.n.

Shah, Ghate, Paranjape, Kumar, 2017. *Application of Hierarchical Temporal Memory Theory for Document Categorization*. s.l., s.n.

Shah, Ghate, Paranjape, Kumar, 2018. *Application of Hierarchical Temporal Memory - Theory for Document Categorization*. San Francisco, s.n., pp. 1-7.

Shepherd, 2011. The Microcircuit Concept Applied to Cortical Evolution: from Three-Layer to Six-Layer Cortex. *Frontiers in Neuroanatomy,* Volume 5 30.

Sherstinsky, A., 2020. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *Physica D: Nonlinear Phenomena,* Volume 404.

Snyder, A., 1986. Encapsulation and inheritance in object-oriented programming languages. *ACM Digital Library,* pp. 38-45.

Solomon D, Henri, Charles, Melwin, Ninoshka, 2015. Neuron the Memory Unit of the Brain. *Neuron the Memory Unit of the Brain,* 17(4), pp. 48-61.

Solomon, Henry, Sushmitha, Partha, Melwin, 2015. *Neuron the Memory Unit of the Brain.* [Online]
Available at: http://www.iosrjournals.org/iosr-jce/papers/Vol17-issue4/Version-3/G017434861.pdf

Sousa, Lima, Abelha, Machado, 2021. Hierarchical Temporal Memory Theory Approach to Stock. *Electronics,* 14(10).

Spruston, N., 2008. Pyramidal neurons: dendritic structure and synaptic integration. *Nature Review Neuroscience,* Volume 9, pp. 206-21.

Subutai, Hawkins, 2016. How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. *ResearchGate,* pp. 1-17.

Swanson, Maffei, 2019. From Hiring to Firing: Activation of Inhibitory Neurons and Their Recruitment in Behavior. *Frontiers in Molecular Neurosciences,* Volume 12, p. 168.

Thomson, B., 2003. *Interlaminar connections in the neocortex.* [Online]
Available at: https://www.ncbi.nlm.nih.gov/pubmed/12466210

Thronton, Srbic, Main, Chitsaz, 2011. *Augmented Spatial Pooling.* Perth, Australia, s.n.

Tien and Kerschensteiner, 2018. Homeostatic plasticity in neural development. *ND - Neural Development,* p. 13/9.

Ts'o, Zarella, Burkitt, 2009. Whither the hypercolumn. *Journal of Psyhology,* p. 2791–2805.

Turrigiano, Nelson, 2004. Homeostatic plasticity in the developing nervous system. *Nature Reviews Neuroscience,* pp. 97-107.

Turrigiano, Nelson, 2004. Homeostatic plasticity in the developing nervous system. *Nature Reviews Neuroscience,* p. 97–107.

Wagner, Anderson, Kulikov, 2022. *Lock Statement.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock

Weliky, Fiser, Hunt, Wagner, 2003. Coding of Natural Scenes in Primary Visual Cortex. *Neuron,* Volume 37, pp. 703-18.

Wheeler, D. A., 2004. Secure programmer: Prevent race conditions. *IBM developerWorks.*

Wielgosz, Pietron, Wiatr, 2016. Using Spatial Pooler and Hierarchical Temporal Memory for object classification in noisy video streams. *Frontiers in Neural Circuits,* Volume 10, p. 13.

Wielgosz, Pietroń, Wiatr, 2016. Using Spatial Pooler of Hierarchical Temporal Memory for object classification in noisy video streams. *IEEE Xplore,* pp. 271-274.

Williams, S. R., 2005. Encoding and Decoding of Dendritic Excitation during Active States in Pyramidal Neurons. *JNeorosci,* 25(25), pp. 5894-5902.

Wilson, G., 2007. *Beautiful concurrency.* s.l.:O'Reilly.

Xinogalos, S., 2015. *Object-Oriented Design and Programming: An Investigation of Novices' Conceptions on Objects and Classes.* [Online]
Available at: https://www.researchgate.net/publication/280554743_Object-Oriented_Design_and_Programming_An_Investigation_of_Novices'_Conceptions_on_Objects_and_Classes

Yang, Kent, Aubanel, Taylor, 2015. *A monitor-based synchronization approach for Java packed objects.* s.l., s.n., pp. 192-200.

Yuwei, Ahmad, Hawkins, 2017. A Theory of How Columns in the Neocortex Enable Learning the Structure of the World. *Frontiers in Neural Circuits,* Volume 11, p. 81.

Zito, Svoboda, 2002. Activity-dependent synaptogenesis in the adult Mammalian cortex. *Neuron,* Volume 35, p. 1015–1017.