DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Using Neurobiological Frameworks for Anomaly Detection in System Log Streams

GUSTAF RYDHOLM

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

KTH Electrical Engineering

Using Neurobiological Frameworks for Anomaly Detection in System Log Streams

GUSTAF RYDHOLM

Examiner: Joakim Jaldén
Academic Supervisor: Arun Venkitaraman
Industrial Supervisor at Ericsson: Armin Catovic

Master's Thesis in Signal Processing
School of Electrical Engineering and Computer Science
Royal Institute of Technology, SE-100 44 Stockholm, Sweden
Stockholm, Sweden 2018

Abstract

Artificial Intelligence (AI) has shown enormous potential, and is predicted to be a prosperous field that will likely revolutionise entire industries and bring forth a new industrial era. However, most of today's AI is either, as in deep learning, an oversimplified abstraction of how an actual mammalian brains neural network function, or methods sprung from mathematics. But, with the foundation of the bold ideas of Vernon Mountcastle stated in 1978 about the neocortical functionality, new frameworks for creating true machine intelligence have been developed, and continues to be.

In this thesis, we study one such theory, called *Hierarchical Temporal Memory* (HTM). We use this framework to build a machine learning model in order to solve the task of detecting and classifying anomalies in system logs belonging to Ericsson's component based architecture applications. The results are then compared to an existing classifier, called *Linnaeus*, which uses classical machine learning methods. The HTM model is able to show promising capabilities of classifying system log sequences with similar results compared with the Linnaeus model. The HTM model is an appealing alternative, due to the limited need of computational resources and the algorithms ability to effectively learn with *"one-shot learning"*.

Referat

Anomali Detektion i System Loggar med Hjälp av Neurobiologiskt Ramverk

Artificiell Intelligens (AI) har visat enorm potential och är förutspådd att revolutionera hela industrier och introducera en ny industriell era. Men, mestadelen av dagens AI är antingen optimerings algoritmer, eller som med deep learning, en grovt förenklad abstraktion av däggdjurshjärnans funktionalitet. År 1978 föreslog dock Vernon Mountcastle en ny järv idé om hjärnbarken funktionalitet. Dessa idéer har i sin tur varit en inspiration för teorier om sann maskinintelligens.

I detta examensarbete studerar vi en sådan teori, kallad *Hierarchical Temporal Memory* (HTM). Detta ramverk använder vi sedan för att bygga en maskininlärningsmodel, som kan hitta och klassificera fel och icke-fel i systemloggar från komponent baserad mjukvara utvecklad av Ericsson. Vi jämför sedan resultaten med en existerade maskininlärningsmodel, kallad *Linnaeus*, som använder sig av klassiska maskininlärningsmetoder. HTM-modellen visar lovande resultat, där HTM-modellen klassificera systemloggar korrekt med snarlika resultat som Linnaeus. HTM-modellen anses vara en lovade algoritm för framtida evalueringar på ny data då den för det första kan lära sig via *"one-shot learning"*, och för det andra inte är beräkningstung modell.

Acknowledgement

First and foremost, I would like to thank my industrial supervisor, Armin Catovic, at Ericsson for giving me the opportunity of exploring the fascinating theories of the neocortex, the support during the project, and the interesting discussions we had.

I'm also grateful to my supervisor Arun Venkitaraman and examiner Joakim Jaldén for reviewing this work.

My journey at KTH started with a preparatory year completely unknowing of what was about to come, but with a determined mind. In the first week or so I was lucky enough to find a group of people to study with. We shared the hardships of the first physics course, hardly being able to draw out the forces acting on a skateboarder in a textbook example correctly. But as we shared these challenging times together our friendships grew, and these people became and will forever be some of my best friends, with whom I shared some of my best times with. So thank you Alan, Carl-Johan, Christian, and Simon.

I would like to especially thank Joakim Lilliesköld for convincing me and my friends into selecting the electrical engineering programme. During my time as a student of electricity I met many lifelong friends in our chapter hall, Tolvan. To them I owe a heartfelt thank you, as they made the life as an engineering student more enjoyable.

To my siblings; Daniel, Sara, Johan, and Hanna, you all are true sources of inspiration, without you I would not be where I am today.

Lastly, to my wonderful parents, I can not express in words how much your support and belief in me has meant. Thank you for always being there for me; for the unconditional love and support.

Gustaf Rydholm,
Stockholm, 2018

Contents

List of Figures

List of Tables

Abbreviations

AI Artificial Intelligence

ANN Artificial Neural Network

CBA Component Based Architecture

GABA gamma-Aminobutyric acid

HTM Hierarchical Temporal Memory

Neocortex New Cerebral Cortex

NMDA N-methyl-D-aspartate

NuPIC Numenta Platform for Intelligent Computing

SDR Sparse Distributed Representations

Chapter 1

Introduction

We start by giving an introduction to the purpose of this thesis and previous and related work. Then, we give the reason for using the entire data set for both training and testing. Finally, we briefly go over the main topics of each of the following chapters.

1.1 Background

Recent techniques in machine learning have shown impressive results when used on large scale data in order to find and classify patterns, most commonly in the field of image, speech and text classification [1]–[3]. At the Ericsson System & Technology unit, a fault detection system has been developed that is able to predict faults from system logs generated by Ericsson's *Component Based Architecture* (CBA) applications. The fault detection system uses traditional machine learning techniques of logistic regression trained with stochastic gradient descent [4]. However, the System & Technology unit wanted to explore new ways to improve their fault detection system by reimplementing it with a state-of-the-art machine learning algorithm called *Hierarchical Temporal Memory* (HTM). HTM, unlike most machine learning algorithms, is a biologically derived algorithm from research done in neuroanatomy and neurophysiology. The intent of HTM theory is to replicate the information processing algorithm in the *new cerebral cortex* (neocortex) [5]. The theory and framework behind HTM is open-source and has been developed by a company called Numenta.

The reason why Ericsson is looking to solve this classification problem

with machine learning, is because the system logs are changing during the development of the CBA application, e.g. components are upgraded, or new ones are installed. Therefore, a basic template matching of system logs with known faults would not work, as Ericsson discovered when they investigated this before. Thus, there is a need for a machine learning application that is able to detect semantically similar logs, while at the same time being robust to changes.

1.2 Purpose and goals

In this degree project we explore the possibility of anomaly detection in system logs with an HTM model. Therefore, the thesis main objective is to do a proof of concept, where we compare the HTM models performance with the existing model. The HTM model works by predicting a fault class for each system log it is fed. The reason for this model behaviour is to quickly go through system log files and detect if there are any faults or anomalies present. The model tries to solve the problem of the long lead times that exist today between fault, detection, and repair. If the model is reliable, it would give a technician the advantage of having a narrower search space when troubleshooting a faulty application.

1.3 Related Work

Linnaeus is a fault detection system that detects and classifies faults in Ericsson's CBA applications. The main reason for its development was to reduce the lead time from the moment a fault is detected until it is repaired. Previously, vast amounts of time was "wasted" by going through the system logs manually and passing them between engineering teams before the faulty component was identified and addressed. The intention of Linnaeus is to perform real-time classification and fault prediction on production systems. Linnaeus is beneficial in two ways; firstly, it raise an alarm if there is a potentially fault. Secondly, it could help a technician with information when writing a trouble report and sending it to the correct software component team straight away [4].

To classify data, Linnaeus uses supervised learning. Hence, a labeled training set is needed to train the parameters of the model. The raw data is acquired from CBA applications running in test environments and collected

in a database [4]. The data is fed through a data preparation pipeline where it is down sampled to only contain labeled examples of relevant logs of faults and non-faults. This is then saved as the training set [4].

To train the model, the training data is fed through a training pipeline, where the parameters of the machine learning model are optimised via a learning algorithm. The training data is transformed by only keeping whole words, and removing unnecessary specific data such as component names and time stamps. The words in the system log file are segmented either using uni-grams or bi-grams, thus individual words or two consecutive words together can be captured. The importance of the uni- and bi-grams are extracted by using Term-Frequency Inverse Document Frequency, which transforms the data into a matrix, where each word is represented by a value of its importance. This input matrix is then used to train a logistic regression model, where the weights of the model are trained with stochastic gradient descent, and the optimal hyper-parameters are found via grid search [4]. The classification model is then saved for later use in the classification pipeline, where real-time system log files are fed to the model for classification [4].

1.4 Research Question

Due to the proof of concept nature of the task of the degree project, we investigate research problems which are more comparative in nature. Therefore, we will in this thesis look at the advantages and disadvantages of using a HTM model for fault detection of system logs, and compare these findings with the machine learning model of the existing system.

1.5 Limitations

The data set was not split into a training and testing set. The entire data set was used in both training and testing of the model. This decision was made due the unequal distribution of examples in the data set, where 95.2% of the total 14459 examples belongs to one class. To still get a sense of the capabilities of the HTM model, one-shot learning was used, which means that each system log was only presented once to the model during training.

1.6 Outline

The thesis is structured into the following sections; theory, method, results, and discussion and conclusion. In chapter 2, we give a brief introduction of component based architecture and system logs, to give context to the data set. Then, in chapter 3, we give a detailed explanation of the HTM algorithm. In chapter 4, we introduce the encoding method of system logs into binary semantic input vectors. The preprocessing step of the data set, together with the HTM model is presented in chapter 5. An evaluation of the HTM models performance and comparison with *Linnaeus* is given in chapter 6. In chapter 7 we have the discussion and conclusion of the thesis. Finally, we have Appendix A, which aims to give an overview of the neuroscience of the neocortex, to better grasp HTM theory.

Chapter 2

Component Based Architecture

In this chapter we introduce the notion of *Component Based Architecture* (CBA), in order to give context to the data set and problem we aim to solve with machine learning. First of all, we briefly cover the motivation for developing CBA application. Then, we proceed to the definition of a software component and how they are connected to create a functional software application. Finally, we introduce system logs and the structure of these.

2.1 Overview and Software Design

CBA has turned into an appealing software design paradigm for software oriented corporations, such as Ericsson. This is due to two things; firstly, it is a way for a company to shorten the development time for their software applications. Secondly, CBA makes it easier to manage the software as components are designed with reusability and replacement in mind [6].

2.1.1 Components

A software component is a modular software object that encapsulates some predefined functionality in the overall set functionality provided by the entire CBA application. To better understand a software component we use the definition of the three properties that defines a component according to Szyperski et al. [7]. First, a component has to be *independently deployable*, which means that its inner functionality is shielded from the outer environment. The property also excludes the ability for a component to

be deployed partially. Secondly, a component has to be a *self-contained entity in a configuration of components*. In order for a component to be self-contained, a specification of requirements has to be clearly stated, together with the services it provides. This is achieved by having well-defined interfaces. An interface acts as an access point to the inner functionality, where the components are able to send and receive instructions or data. A component usually have multiple interfaces, where different interfaces allows clients, i.e. other components or systems, to invoke different services provided. However, to properly function, components also have to specify their dependencies, which define rules for the deployment, installation, and activation of the specific component. We illustrate the first two properties with an abstract CBA application in Figure 2.1. Finally, a component can not have *externally observable states*, which means that internal states can be cached or erased, without major consequences to the overall application.

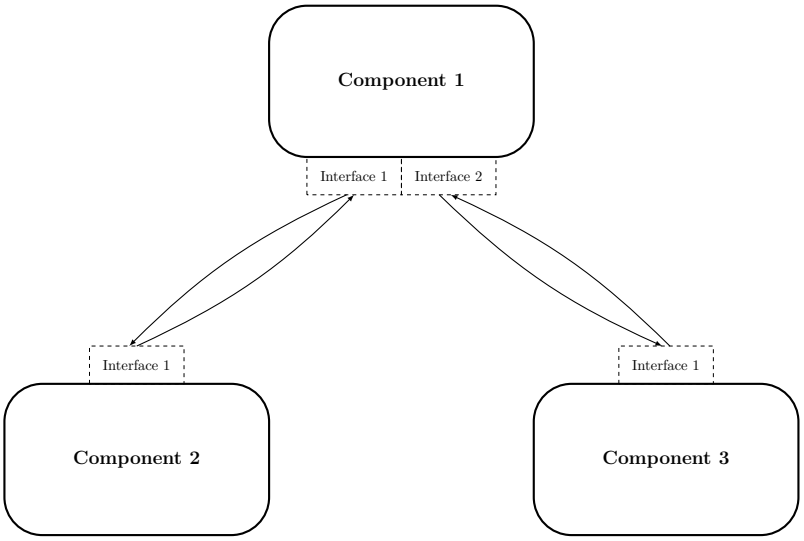

Figure 2.1. An abstract illustration of a CBA application of three components interacting via their interfaces.

2.1.2 Challenges in Development of CBA Applications

A CBA application consist of a topology of interconnected components, as shown in Figure 2.1, where each component is responsible for some part

of the overall system functionality. Individual components are built using traditional software engineering principles, which means that they are developed and tested in isolation before full-scale system integration and testing. Thus, errors usually occur due to the composition of the CBA application, where unforeseen states of component can cause them to fail [7]. To find out which components failed during system integration, a technician usually has to go through the system logs generated during the testing in order to find the cause of failure. As this is a time consuming work for a technician, the aim is to automate this task. To get a better understanding of the challenges when trying to automate this task we now proceed to go through the structure of the system logs generated in the test suite environment.

2.2 System Logs

System logs are generated in test suites, where all events executed are recorded in system log files. The structure of system logs we analyse in this thesis is presented in Figure 2.2.

```
Time_stamp Log_location process[PID]: Description
```

Figure 2.2. The general structure of a system log.

To understand each term of the system log we will go through them one by one.

- `Time_stamp` – At which time the event was executed/logged.
- `Log_location` – The system log will either be logged in a *system controller* or in a *system Payload*, denoted *SC* and *PL* respectively.
- `process` – The name of the process that executed the command.
- `[PID]:` – The process identification, surrounded by brackets and ending with a colon.
- `Description` – States what occurred or action taken, and in some cases the severity level of the event.

The first three terms of the system log are rather self explanatory and thus we will not go into them any further. However, description is the term that is most important to solving the task of making the analysis of system logs autonomous. The description include information such as, severity, e.g. *error* or *rebooting*, effected components and their states, and sometimes a reason is stated. The challenge with the description is however that they do not follow a set of predefined rules, and are generally ill-structured. The reason for this is that each system log is written by a specific software engineering team who developed that component. Each software team or developer has their own unique style and selection of acronyms that becomes troublesome. Due to format and inconsistency of the description, it will impact the ability of the machine learning models to learn the patterns that separate the individual fault classes in the system logs. We have now introduced the general structure of the data set and we will now proceed to the theory of the machine learning model we will use.

To summarise this chapter; we started off by introducing the reason for component based software applications. Then, we briefly defined what a software component is and how it interacts with its environment. Next, we proceeded to state the challenges in the development of CBA systems. Finally, the general structure of the system logs was stated to get a context to the data set and its limitations.

Chapter 3

A Brain-Inspired Algorithm

Hierarchical Temporal Memory (HTM) is a theory of how the human neocortex processes information, be it auditory, visual, motor, or somatosensory signals, developed by Jeff Hawkins [8]. It was long known that the neocortex looked remarkably similar in all different regions, this made anatomists to focus on the minuscule differences between the cortical regions to find the answer how they worked. However, in 1978, a neuroscientist by the name of Vernon B. Mountcaslte stopped focusing on the differences and saw that it was the similarities of the cortical regions that mattered. He came up with a theory of a single algorithm by which the neocortex processes information. The small differences that the anatomists found, was only because of the different types of signals that where being processed, not a difference in the algorithm itself [8]. With this idea of a single algorithm, Jeff Hawkins started a company called Numenta, with the intention of figuring out this algorithm. Numenta has since published an open-source project named Numenta Platform for Intelligent Computing, or NuPIC, which is an implementation of the HTM theory.

In this chapter, we will start by introducing the HTM neuron. Then, we proceed to explain the way information is represented in an HTM network. Finally, we explain the sequence learning algorithm, called temporal memory.

3.1 HTM Neuron Model

The HTM neuron model is an abstraction of the pyramidal neuron in the neocortex [9]. However, so is the point neuron, which is used in most *Artificial Neural Networks* (ANNs); so how different are they? The point neuron, see Figure 3.1, summates all the inputs on its synapses, then passes this value through a non-linear activation function. If the output value is above a threshold, the neuron outputs the value of the activation function; otherwise it will output a zero. With the properties of the dendrites explained in section A.2, one could argue that point neurons does not have dendrites at all, and therefore completely lack the active properties of the dendrites.

The connection between the point neurons is instead the synaptic connection, which can be changed via back propagation [10].

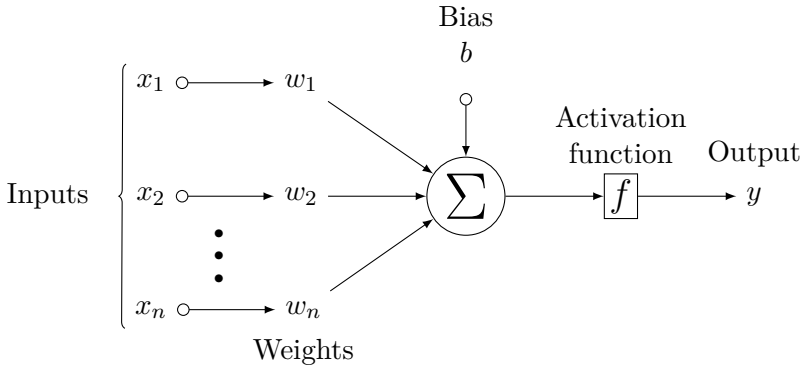

Figure 3.1. The point neuron, used in most ANNs, summates the synaptic input and passes in through an activation function. It lacks active dendrites and only has synaptic connections.

The HTM neuron, see Figure 3.2, is modelled on the active dendrite properties, and is therefore able to make use of the coincidence detection of the pyramidal neuron. The coincidence detection is activated via non-linear integration when a small number of synapses, experimentally show to be around 8-20, are active in close spatial proximity on a dendritic segment [9]. This non-linear integration will cause an NMDA dendritic spike, thus allowing the neuron to recognise a pattern [9]. In order for the neuron to be able to recognise a vast number of different patterns, the active input pattern needs to be sparse, i.e. only a few neurons that are active per input pattern.

If we assume that the total number of neurons in a population is n, and at any given time the number of active cells are a, then sparse activation is given as $a \ll n$. On each dendritic segment there are s number of synapses. For a dendritic segment to release an NMDA spike, the number of synapses that needs to be active is θ, i.e. the NMDA spike threshold, of the total number of synapses s [9]. By forming more synaptic connections for each pattern than necessary the neuron becomes more robust to noise and variation in the input pattern. However, the trade-off in these extra connections is the increased likelihood of the neuron to classify false positives, but if the patterns are sparse the increased likelihood is infinitesimal [9]. The dendrites can be divided into three zones of synaptic integration, the *basal*, *proximal*, and *distal zone* [9]. These zones are categorised based on input and spatial position on the neuron, and are explained below.

3.1.1 Proximal Zone

The feedforward input is received by the dendrites in the proximal zone, as this is the main receptive field of HTM neuron [9]. The proximal zone is the dendritic zone closest to the soma, usually consisting of several hundreds of synapses. Because of the proximity to the soma, the NMDA spike generated in this dendritic zone is strong enough to effect the soma in such a way that it generates an action potential. If the input pattern is sparse, subsets of synapses are able to generate NMDA spikes. Therefore, the coincident detector can detect multiple different feedforward patterns in one input signal, thus it can be viewed as a union of several unique patterns [9].

3.1.2 Basal Zone

The basal zone is the dendritic segment that connects neuron in different minicolumns to each other. These connection allow a neuron to detect activity of neighbouring neurons, which enables individual neurons to learn transitions of input patterns. When a basal segment recognises a pattern, it will generate an NMDA spike. But due to the distance from the soma, the signal attenuates and is not able to generate an action potential in the soma. However, it does depolarise the soma, also called the *predictive state* of the neuron. The *predictive state* is an important state of the neuron because it has major contribution to the overall network functionality. If a neuron is in the predictive state it will become active earlier than its neighbours, in

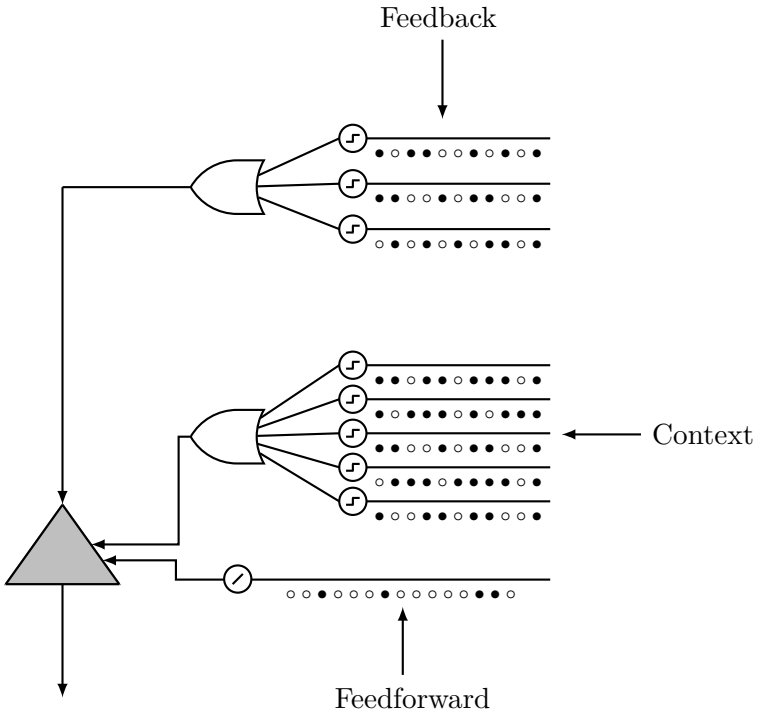

Figure 3.2. The schematic of the HTM neuron with arrays of coincident detectors consisting sets of synapses. However, in this figure only a few is shown, where black dots represents active synapses and white inactive ones. An NMDA spike is generated if the total number of active synapses are above the NMDA spike threshold, θ, (represented as a Heaviside node) on any of the coincident detectors in a dendritic zone, which is represented by OR-gate. The dendritic zones can be divided in to three different zones base on the distance from the soma and synaptic connections. The proximal dendrites receive the *feedforward* pattern, also know as the receptive field of the neuron. The basal zone receives information about the activity of neighbouring neurons of which its connected to and can be seen as giving *context* to the input pattern. Apical dendrites receive the *feedback* information form the layers above which also can effect the state of the soma [9].

the same minicolumn and close proximity, if the feedforward pattern activates the proximal segment. When a neuron transitions from the predictive state to the active state it will not only give of an action potential, but also inhibit its neighbours from becoming active. Thus, keeping the activation

pattern for recognised input patterns sparse [9]. This type of inhibition of nearby neurons is a way to represent the functionality of inhibition neurons, without representing them as individual cells [11].

3.1.3 Distal Zone

Furthest from the soma is the apical dendrites, which connects neurons to the ascending layers. Much like the basal dendrites, apical segment does not generate a signal strong enough to cause an action potential in the soma. The signal generated on the apical segment differs from the signal generated on the basal segment. When a pattern is recognised, the NMDA spike does not directly travel to the soma. Instead, the soma is depolarises by a calcium ion, Ca^{2+}, spike generated at the dendritic segment. This depolarisation gives the neuron a ability of doing top-down extraction [9].

3.1.4 Learning

The learning of an individual HTM neuron is based on two principles; formation and removal of synaptic connection via Hebbian style learning [9]. Each dendritic branch has a set of potential synaptic connection, where each connection can become active if there is enough simultaneous activity between the two potentially connected neurons. For a dendritic branch to recognise a pattern there needs to be a subset of the connected synapses that are active. This threshold is usually set to 15-20 [9]. When a dendritic branch becomes active, the entire dendritic segment is seen as active to the neuron, which is visualised in Figure 3.2 by the OR gate. The HTM neuron learns to detect new pattern by forming new synaptic connection on a dendritic branch. Each potential synaptic connection is given a *permanence* value, which determines the strength of a synaptic connection between the neuron's dendritic branch and another neuron's axon in the network. The permanence value is defined on the range $[0, 1]$, where 0.0 means that there is no connection, and 1.0 means that a fully formed synapses has been grown. The potentiation and depression of each permanence value is achieved via a Hebbian learning rule, i.e. if neurons fire together they wire together. In order for a synaptic connection to be formed, the permanence value has to be above a certain threshold, e.g. 0.3. With a permanence value above the threshold, the synaptic weight is assigned to 1 between the neurons. Therefore, there is no difference if the permanence value is 0.3 or 1.0 when

a pattern is recognised. However, the lower the value is, the easier it is for the neuron to forget the connection, and in extension the pattern. With this growing mechanism of the neuron, the tolerance to noise and on-line learning is possible [9].

3.2 Information Representation and Detection

3.2.1 Sparse Distributed Representation

The presynaptic input patterns of information that is received by the dendrites, needs to be robust to noise and have a vast encoding capacity, as the neocortex handles an endless stream of sensory input. Empirical evidence shows that the neocortex operates by using sparse representations of information [12]. In HTM theory, these sparse activation patterns are called *Sparse Distributed Representation*, or SDR. Sparseness is due to the low number of active neurons at any given time, and it is distributed as no information is encoded in a single neuron. The information of the cell activity that the dendrite receives from the presynaptic cells is either active or non-active, and can therefore be modelled as a binary vector [12].

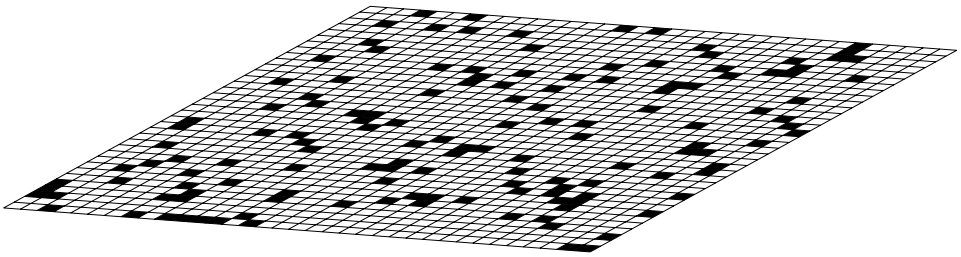

Figure 3.3. An example of an SDR, where black squares represent active cells and white squares represent inactive ones. The SDR is represented as matrix for convenience.

In Figure 3.3, we present an example of presynaptic input, or SDR, represented as a bit vector. Each cell can either be active or inactive, represented as black and white squares respectivly. The entire presynaptic input space of n cells, at time t, for a dendritic segment is represented by an n-dimensional SDR, $\boldsymbol{X}_t$:

$$\boldsymbol{X}_t = [b_0,\ b_1, ...,\ b_{n-1}] \tag{3.1}$$

where $b_i \in \mathbb{Z}_2$. The number of active cells is given by $w_X = |\boldsymbol{X}_t|$ and the pattern is considered sparse if $w_X \ll n$. The number of possible encodings the presynaptic input pattern is given by:

$$\binom{n}{w_X} = \frac{n!}{w_X!(n - w_X)!} \tag{3.2}$$

3.2.2 Presynaptic Pattern Detection

A dendritic segment can also be modelled as a binary vector $\boldsymbol{D}$ of length n, which represents both potential and established synaptic connections to the presynaptic input. The active synaptic connections are represented as the non-zero elements, where b_i is the synaptic connection to the presynaptic cell i. The number of established connections, represented by $s = |\boldsymbol{D}|$, has been experimentally shown to typically be around 20 to 300 synapses for a dendritic segment. But the potential connections can be in the thousands [12]. To generate a spike in a dendritic segment, the number of synaptic connections that receive active input form the presynaptic cells needs to exceed a threshold, θ. This threshold can be computed via the dot product of the two binary vectors:

$$m(\boldsymbol{X}_t, \boldsymbol{D}) \equiv \boldsymbol{X}_t \cdot \boldsymbol{D} \geq \theta \tag{3.3}$$

Where the threshold usually is lower than the number of connections and presynaptic activity, i.e. $\theta \leq s$ and $\theta \leq w_X$. As $\boldsymbol{X}_t$ does not represent the full presynaptic input pattern, but only a subsample, each dendritic segment only learns synaptic connections to some active cells of the entire pattern. How well a dendritic segment detects a pattern is dependent on the value of the NMDA spike threshold, θ, and the robustness to noise in SDR encodings. With lower values of θ the dendritic branch is able to detect known input patterns easier. However, there is an inherent trade-off in small values of θ, as the dendritic branch are then more likely to detect false positives if there is noise in the input pattern [12].

If θ is set to a reasonable value, e.g. around 8-20, the probability of false detection due to noise in the SDR will be extremely unlikely. The reason for this is the sheer size of possible combination of ON-bits in the SDR. SDRs corrupted by some noise will not overlap enough to been interpreted as another possible input pattern. Therefore, detection on each dendritic

segment with inexact matching has a very low probability of false detection, as described by Ahmad et al. in [12].

In each cortical region there are millions of neurons that simultaneously trying to recognise hundreds of patterns. They are able to recognise hundreds of patterns, as there only needs to be between 8 to 20 active synapses to generate an NMDA spike. On each of these neurons there are numerous dendritic branches, that combined have several thousands of synapses on them. Therefore, the robustness of the single dendritic segment needs to be maintained throughout a large neuron network [12].

To quantify the robustness properties in the a larger scale we will first introduce the probability of false positives for an arbitrary number of dendritic segments, of which do not have to belong to the same neuron. Let M be the number of different patterns represented by M different dendritic segments, all of which has the threshold θ and s number of synapses. The set of the dendritic segments is given by $S = \{\boldsymbol{D}_0,\ \boldsymbol{D}_1,\ ...,\ \boldsymbol{D}_{M-1}\}$, where $\boldsymbol{D}_i$ represents a dendritic segment vector. Let $\boldsymbol{X}_t$ be a random presynaptic input, of which is classified as belonging to the set if the following is true:

$$\boldsymbol{X}_t \in S \ := \ \exists_{\boldsymbol{D}_i}\ m(\boldsymbol{D}_i, \boldsymbol{X}_t) = True \tag{3.4}$$

There is no false negatives if the number of corrupt bits in $\boldsymbol{X}_t$ is $\leq w_{D_i} - \theta$. The probability of a false positive is given by:

$$P(\boldsymbol{X}_t \in S) = 1 - (1 - P(m(\boldsymbol{X}_t, \boldsymbol{D}_i)))^M \tag{3.5}$$

Which computationally difficult to compute as the probability of individual overlap is extremely unlikely [12].

3.2.3 The Union Property on Dendritic Segments

Another important property that comes with the SDR encoding is the ability to group and reliably store a set of SDR with a single SDR representation. This is achieved by taking the union of all the SDRs in a set, and is called the *union property* [12]. For binary vectors this is equivalent to taking the Boolean OR between all vectors. The ability to store multiple patterns is an important feature of the dendritic segment. In dendritic segments the synapses that respond to different patterns are stored in the same SDR. Thus, multiple presynaptic patterns can cause an NMDA spike to be generated. For a dendritic segment, $\boldsymbol{D}$, to be able to detect an arbitrary number

of M synaptic SDRs, we simply take union of all individual synaptic connection vectors, $\boldsymbol{d}_i$:

$$\boldsymbol{D} = \bigcup_{i=0}^{M-1} \boldsymbol{d}_i \tag{3.6}$$

The patterns will be detected as long as θ number of synapses are active. By increasing the M, the number of patterns that a dendritic segment can detect increases, but so does the probability of false detection. Therefore, there is a limit to the amount of ON-bits a dendritic segment can have, before the detection of false positives becomes a problem [13]. Using unions, we are able to make temporal predictions, temporal pooling, create invariant representations, and create an effective hierarchy [13].

3.3 Temporal Memory

3.3.1 Notation

The sequence learning algorithm of the HTM theory is called the *temporal memory* [9]. The temporal memory consists of a layer of N mini-columns stacked vertically. Each mini-column contains M number of HTM neurons, thus a total of NM cells. The cells can be in one of three states; active, non-active, or predictive (depolarised). Thus, for a given time-step, t, the active cells in the layer are represented by the $M \times N$ binary matrix, $\boldsymbol{A}^t$, where a_{ij}^t is the current active (non-active) state of the i'th cell in the j'th minicolumn [9]. For the same time-step, the predictive state of each cell is given by the $M \times N$ binary matrix, $\prod^t$, of which the predictive state of the i'th cell in j'th minicolumn is denoted by π_{ij}^t [9].

Each cell in a layer has the potential to connect to any other cell via its basal dendrites. The set of basal dendritic segments of the i'th cell in j'th minicolumn is therefore represented by $\boldsymbol{D}_{ij}$. Each segment has a subset of s potential synaptic connections from the $NM - 1$ cells in the layer. This subset is associated with a non-zero permanence value, where the d'th dendritic segment is represented as a $M \times N$ sparse matrix, $\boldsymbol{D}_{ij}^d$. A synaptic connection is only considered to be established if the permanence value is above a certain threshold. To represent these synapses with a weight of 1 on the same dendritic segment, we have the following $M \times N$ binary matrix, $\tilde{\boldsymbol{D}}_{ij}^d$ [9].

3.3.2 Initialisation of the Dendritic Segments

With the initialisation of the network, each cell's dendritic segments are randomly assigned unique sets of s potential synaptic connections. The non-zero permanence value of these connections is randomly initialised, with some being above the threshold and thus being connected, while others are not and therefore are unconnected [9].

3.3.3 Activation of Cells

Each minicolumns feedforward receptive field is a subset of the entire feedforward pattern [9]. The receptive field of a minicolumn is shared by all cells in that minicolumn. A minicolumn becomes active if the number of synapses connected to the receptive field is above a certain threshold. However, there is an upper bound of k minicolumns that can be active at the same time. Thus the minicolumns that have the highest number of active synapses get selected, which is also called the inhibitory process [9]. The set of k winning minicolumns is denoted by $\boldsymbol{W}^t$. The active state of the individual cells in each minicolumn is computed by:

$$a_{ij}^t = \begin{cases} 1, & \text{if } j \in \boldsymbol{W}^t \text{ and } \pi_{ij}^{t-1} = 1 \\ 1, & \text{if } j \in \boldsymbol{W}^t \text{ and } \sum_i \pi_{ij}^{t-1} = 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

In the first case the cell will become active if it was in a predictive state in the time-step before. In the second case, all cells in a minicolumn will become active if none of them previously where in a predictive state. If none of these cases applies, the cell will remain inactive [9]. Next, the predictive state of each cell in the winning column is computed as follows:

$$\pi_{ij}^t = \begin{cases} 1, & \text{if } \exists_d \left\| \tilde{\boldsymbol{D}}_{ij}^d \circ \boldsymbol{A}^t \right\|_1 > \theta \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

For a cell to become depolarised in the current time-step, the contextual information received from the presynaptic input on any basal dendritic segment needs to be above the NMDA spike threshold, θ. In order to detect if a segment is above this threshold, an element-wise multiplication, represented by $\circ$, of the dendritic segment and the active cells in the layer is computed.

The L_1-norm of the result is then computed and compared with the threshold. In order for a cell to become depolarise, at least one segment needs to be active [9].

3.3.4 Learning of Dendritic Segments

The reason a layer is able to learn multiple functionalities is due to the plasticity of the synapses belonging the cells [9]. In the HTM neuron, the updating rule for the permanence value of the synapses is a Hebbian-like rule. That is, if a cell was in a predictive state in a previous time-step, and then becomes active in the current because of the feedforward pattern, the synaptic connection that cause the depolarisation gets reinforced [9]. The segments responsible for the depolarisation are selected via the following operation:

$$\forall_{j \in \boldsymbol{W}^t} \left(\pi_{ij}^{t-1} > 0 \right) \text{ and } \left\| \tilde{\boldsymbol{D}}_{ij}^d \circ \boldsymbol{A}^{t-1} \right\|_1 > \theta \tag{3.9}$$

First, the winning columns that had cells in a predictive state is selected. Next, the dendritic segments of these cells that cause the depolarisation is selected. However, if a winning column did not have cells in a predicted state, we need to reinforce the connection of the cell that had the most active segment. As this allows the cell to represent the transition of the sequence if it repeats later on [9]. To select the segment that where the most active, we first denote $\dot{\boldsymbol{D}}_{ij}^d$ as the $M \times N$ binary matrix of $\boldsymbol{D}_{ij}^d$, where each positive permanence value is represented as a 1 [9]. Next, we select the winning columns that did not have a cell in a predictive state, and then take the cell with the most active dendritic segment in each minicolumn.

$$\forall_{j \in \boldsymbol{W}^t} \left(\sum_i \pi_{ij}^{t-1} = 0 \right) \text{ and } \left\| \tilde{\boldsymbol{D}}_{ij}^d \circ \boldsymbol{A}^{t-1} \right\|_1 = max_i \left(\left\| \dot{\boldsymbol{D}}_{ij}^d \circ \boldsymbol{A}^{t-1} \right\|_1 \right) \tag{3.10}$$

With the ability to select the relevant segments that cause a cell to become active, we now need to define the Hebbian-like learning rule, i.e. wire together fire together. That is, we reward connections with active presynaptic input, and punish the synapses that does not. To achieve this we decrease all permanence values by a small value p^-, while at the same

time rewarding the connection with active presynaptic input by increasing them with a larger value p^+ [9].

$$\Delta \boldsymbol{D}_{ij}^d = p^+ \left(\dot{\boldsymbol{D}}_{ij}^d \circ \boldsymbol{A}^t \right) - p^- \dot{\boldsymbol{D}}_{ij}^d \tag{3.11}$$

As Equation 3.11 only updates cells that became active, or closest to being active, selected by Equation 3.9 and Equation 3.10 respectively, we need to define an equation for penalising the cells that did not become active [9]. The permanence values of these inactive cells will start to decay with a small value of p^{--}:

$$\Delta \boldsymbol{D}_{ij}^d = p^{--} \dot{\boldsymbol{D}}_{ij}^d \text{ where } a_{ij}^t = 0 \text{ and } \left\| \tilde{\boldsymbol{D}}_{ij}^d \circ \boldsymbol{A}^t \right\|_1 > \theta \tag{3.12}$$

Each cell is then updated with new permanence values for each of its dendritic segments by applying the following update rule:

$$\boldsymbol{D}_{ij}^d = \boldsymbol{D}_{ij}^d + \Delta \boldsymbol{D}_{ij}^d \tag{3.13}$$

As we have now gone through the temporal learning algorithm, we will now summarise this chapter. First, we introduced the HTM neuron and how it is modelled differently from the point neuron used in most ANNs. We then explained the properties of each dendritic segment of the HTM neuron. Then, we moved on to explaining how an HTM neuron learns to recognise input patterns. Next, we introduced the notion of sparse distributed representations (SDRs), which are binary vectors that represent information in an HTM network. With the familiarity of SDRs, we then went over the active processing property of the dendritic segment, and how each segment can detect multiple input patterns, i.e. with the *union property*. Finally, we went over the algorithm for recognising temporal sequences in an HTM network and how it is able to learn to recognise new input patterns.

Chapter 4

The Path to Semantic Understanding

In this chapter we will describe how we encode words into *Sparse Distributed Representation* (SDR). To encode words into SDRs, each bit has to have a unique semantic meaning. To achieve this, we will first transform each word into a numerical vector via the *Global Vectors for Word Representation*, or GloVe, algorithm. Next, each numerical vector is transformed into a binary vector, or SDR, where only the most important elements are converted into ON-bits. Finally, we explain the decoding algorithm for word SDRs to recall the word it encodes.

4.1 Global Vectors for Word Representation

Unlike most semantic vector space models, which are evaluated based on a distance metric, such as distance or angle, the GloVe algorithm tries to capture the finer structures of differences in the word vectors with distributed representation and multi-clustering, and thus capturing analogies [14]. This means that *"king is to queen as man is to woman"* is encoded as the vector equation $king - queen = man - woman$ in the word vector space [14]. This is possible as the GloVe algorithm creates linear directions of meaning with a global log-bilinear regression model [14].

4.1.1 Notation

GloVe is an unsupervised learning algorithm that looks at the co-occurrence of words of a global corpus to obtain statistics for each word, and thus create a vector representation of each word [14]. The word-by-word co-occurrence is represented by the matrix $\boldsymbol{X}$, which entries, x_{ij}, are the number of occurrence of i word in the context of word j [14]. Next, we have the total number of times any word is present in the context of word i, defined as $x_i = \sum_k x_{ik}$. Finally, the probability of word j appearing in the context of word i, $P_{ij} = P(j|i) = x_{ij}/x_i$.

4.1.2 Word to Vector Model

To distinguish between relevant and irrelevant words, and discriminate between relevant word, Pennington et al. [14] found that using a ratio between co-occurrence probabilities was a better option. Thus, formalising the general objective function as:

$$F(\boldsymbol{w}_i, \boldsymbol{w}_j, \tilde{\boldsymbol{w}}_k) = \frac{P_{ik}}{P_{jk}} \tag{4.1}$$

where the ratio depends on three words; i,j, and k. Each word is represented as a word vector $\boldsymbol{w} \in \mathbb{R}^n$, and $\tilde{\boldsymbol{w}} \in \mathbb{R}^n$ is separate context vector. By modifying Equation 4.1 with consideration to the linear structure of the vector space together with symmetrical properties, Pennington et al. [14] is able to find a drastically simplified solution on the form:

$$\boldsymbol{w}_i^\top \tilde{\boldsymbol{w}}_k + b_i + \tilde{b}_k = \log(x_{ik}) \tag{4.2}$$

Where the bias terms b_i and $\tilde{b}_k$ takes care of symmetry issues in x_i and $\tilde{x}_k$ respectively [14]. However, Equation 4.2 is ill-defined as it diverges if the logarithms argument is zero. Another problem is that it weights all co-occurrences equally. Therefore, a weighted least squares regression model is proposed to solve this [14]. By introducing a weighting function $f(x_{ij})$ the following cost function is introduced:

$$J = \sum_{i,j=1}^{V} f(x_{ij}) \Big(\boldsymbol{w}_i^\top \tilde{\boldsymbol{w}}_j + b_i + \tilde{b}_j - \log(x_{ij}) \Big)^2 \tag{4.3}$$

where the vocabulary size is given by V. To address the problems that the weighting function needs to solve, the following criteria are put on the weighting function:

i. First, $f(x)$ should be a continuous function where $f(0) = 0$. It should also fulfil that $\lim_{x \to 0} f(x) \log^2(x)$ is finite.

ii. $f(x)$ must be a monotonically increasing function, as rare co-occurrences are weighted as less important.

iii. To not overweight frequent co-occurrences, i.e. for large values of x, $f(x)$ should be relatively small.

With these requirements, Pennington et al. [14] finds that a suitable weight function is given by:

$$f(x) = \begin{cases} (x/x_{max})^{\alpha}, & \text{if } x < x_{max} \\ 1, & \text{otherwise} \end{cases} \tag{4.4}$$

Where empirical results shows that $\alpha = 3/4$ gives a modest improvement over the linear choice of $\alpha = 1$ [14]. The parameter x_{max} acts as a cut-off, where the weighting function is assigned to 1 for word-word co-occurrences x greater than x_{max}.

4.1.3 Training

The GloVe algorithm is trained on a corpus using *adaptive gradient algorithm* to minimise the cost function. GloVe generates two sets of word vectors, $\boldsymbol{W}'$ and $\tilde{\boldsymbol{W}}$. If $\boldsymbol{X}$ is symmetrical, $\boldsymbol{W}'$ and $\tilde{\boldsymbol{W}}$ only differ due to the randomisation of the initialisation [14]. As explained by Pennington et al. [14], a small boost in performance is typically gained by summating the two word vector sets. Thus, the word vector space generated is defined as $\boldsymbol{W} = \boldsymbol{W}' + \tilde{\boldsymbol{W}}$.

4.2 Semantic Sparse Distributed Representation

For a *Hierarchical Temporal Memory* (HTM) to be able to learn input patterns, each bit of the feedforward receptive field has to encode some unique meaning. In this case, each bit of the SDRs has to encode some unique

semantic meaning. The semantic SDR encoder that we will explain was developed by Wang et al. [15] to convert numerical GloVe vectors into SDR while minimising the semantic loss of the encoding.

4.2.1 Encoding

With word vector matrix $\boldsymbol{W}$ generated by the GloVe algorithm in section 4.1, all words in the corpus now has a unique n-dimensional semantic vector representation.

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \vdots \\ \boldsymbol{w}_V \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{v1} & w_{v2} & \cdots & w_{Vn} \end{pmatrix} \tag{4.5}$$

However, these are numerical vectors and therefore needs to be converted into binary vectors. As only k, where $k \ll n$, elements will be converted into ON bits, we need to be able to find the most semantically important elements of each word vector in order to minimize the semantic loss in the SDR encoding.

We start by introducing what Wang et al. [15] calls the *Bit Importance* metric. This is a scaling operation of each column in the word vector matrix, $\boldsymbol{W}$, defined as:

$$\psi(w_{ib}) = w_{ib} \left(\sum_{j=1}^{V} w_{jb} \right)^{-1} \tag{4.6}$$

where b is seen as the current bit in the SDR encoding and is defined on the range $b \in \{1,\ 2, ...,\ n\}$, and V is the size of the vocabulary of the corpus, i.e. number of word vectors in $\boldsymbol{W}$. This gives a metric of the importance of each element in a word vector. Next, we introduce the *Bit Discrimination* metric, which is a feature standardisation of each word:

$$\phi(w_{ib}) = \left| \frac{w_{ib} - \mu_b}{\sigma_b} \right| \tag{4.7}$$

Which means that we will have a zero-mean and unit variance. The discrimination gives a measure of how important a word vector's element is

in relation to the same element in the other word vectors. The mean used in Equation 4.7 is defined as:

$$\mu_b = \frac{1}{V}\sum_{j=1}^{V} w_{jb} \tag{4.8}$$

where V is the size of the vocabulary. The standard deviation of column in a word matrix is computed in the following way:

$$\sigma_b = \sqrt{\frac{1}{V}\sum_{j=1}^{V}(w_{jb} - \mu_b)^2} \tag{4.9}$$

With element wise multiplication of the i'th bit importance and discrimination vector, the i'th *Bit Score* vector is obtained:

$$\boldsymbol{\Gamma}_i(\boldsymbol{w}_i) = \psi(\boldsymbol{w}_i^\top) \odot \phi(\boldsymbol{w}_i^\top) = \begin{pmatrix} \psi(w_{i1}) \\ \psi(w_{i2}) \\ \vdots \\ \psi(w_{in}) \end{pmatrix} \odot \begin{pmatrix} \phi(w_{i1}) \\ \phi(w_{i2}) \\ \vdots \\ \phi(w_{in}) \end{pmatrix} \tag{4.10}$$

where $\odot$ is the *Hadamard* product. This vector contains a score of how semantically important each element in the word vector $\boldsymbol{w}_i$ is. To encode the bit score vectors to SDR, we need to find the k largest values and convert them to ON-bits, while the remaining $n-k$ values are converted into OFF-bits. The set of the k largest values in $\boldsymbol{\Gamma}_i(\boldsymbol{w}_i)$ is defined as:

$$\boldsymbol{\Gamma}_{i,max}(\boldsymbol{w}_i) = \underset{\boldsymbol{\Gamma}'_i \subset \boldsymbol{\Gamma}_i,\ |\boldsymbol{\Gamma}'_i|=k}{\arg\max} \sum_{\gamma \in \boldsymbol{\Gamma}'_i} \gamma \tag{4.11}$$

where $\gamma = \psi(w_{ij}) \cdot \phi(w_{ij})$. With this set of values, we now convert the i'th word vector $\boldsymbol{w}_i$ into an semantic SDR by applying the following encoding function to each element j:

$$\xi_i(w_{ij}) = \begin{cases} 1, & \text{if } \Gamma_i(w_{ij}) \ \in \ \boldsymbol{\Gamma}_{i,max}(\boldsymbol{w}_i) \\ 0, & \text{otherwise} \end{cases} \tag{4.12}$$

Thus, the SDR encoding $\boldsymbol{x}_i$ of the i'th word is given by:

$$\boldsymbol{X}_i(\boldsymbol{w}_i) = \xi_i(\boldsymbol{w}_i) = \begin{pmatrix} \xi_i(w_{i1}) \\ \xi_i(w_{i2}) \\ \vdots \\ \xi_i(w_{in}) \end{pmatrix} \tag{4.13}$$

Each SDR encoding is represented in an SDR matrix, denoted as $\boldsymbol{X} = [\boldsymbol{X}_1\ \boldsymbol{X}_2\ \cdots\ \boldsymbol{X}_v]^\top$, where the i'th row is the SDR encoding of the i'th word in word vector matrix $\boldsymbol{W}$.

4.2.2 Decoding

From section 3.3 we learned that the temporal memory is able to predict future incoming pattern when some of the neurons transition into depolarised states. The incoming feedforward pattern predicted by the temporal memory is denoted by $\tilde{\boldsymbol{X}}_{t+1}$. To decode the predicted word we simply find the word SDR that share the most ON-bits with the predicted SDR.

$$\tilde{w} = \arg\max_i \boldsymbol{X}_i \cdot \tilde{\boldsymbol{X}}_{t+1} \tag{4.14}$$

where $i \in \{1, 2, \ldots, V\}$. Equation 4.14 returns the index of the best matching word, thus the predicted word can be obtained by doing an index look-up. If the relevant words only belongs to a subset of the entire set of words, the argument i can be changed to only contain these words. Letting the temporal memory being able to only predict fault categories is such a case.

Chapter 5

Method

In this chapter we combine the previous chapters of natural language processing with *Hierarchical Temporal Memory* (HTM) to create an HTM model that is able to classify system logs. First, we explain the preprocessing step of the system logs. After this step, we convert the system logs into *Sparse Distributed Representations* (SDRs) via the encoding process described in section 4.2. Finally, we define the architecture of the HTM network and how it is trained and tested.

5.1 Preprocessing

As with any textual data used for machine learning tasks, a preprocessing step of the system logs is necessary to remove numbers, special characters, and nonessential words. In this preprocessing step a corpus of the entire system log is also created. This is done in order to transform each word into a vector with the *Global Vectors for Word Representation* (GloVe) algorithm.

5.1.1 Data set

The data set consists of thirteen different classes of faults extracted from Ericsson's *Component Based Architecture* (CBA) applications. The system logs was labeled by technicians in trouble reports. The classes together with number of examples are shown in Table 5.1. The data set contains 14459 unique logged events with varying length, from 4 to 206 tokens long. However, the data set is also heavily skewed, with 95.2% of all logged events

Table 5.1. The System log Data Set

Fault Class	Number of examples
CDIA	9
COM	5
CoreMW	74
CoreMWBRF	15
EVIP	16
JavaOAM	8
LDE	37
LM	6
NoFault	14251
SEC	9
SS7CAF	2
Trace	10
VDicosee	17

being labeled as *NoFault*. Some fault classes either had one example or contained single lines of system logs.

Each log is structured in a similar way; date, place of recording, component, and description. An example of a *CoreMW* fault from the data set is:

```
Jul 27 00:18:10 PL-12 osafamfwd[8370]: Rebooting
OpenSAF NodeId = 0 EE Name = No EE Mapped,Reason:
AMF unexpectedly crashed, OwnNodeId = 134159,
SupervisionTime = 60
```

Figure 5.1. An example of a sytem log of a CoreMW fault.

The logs contain unnecessary specifics, such as time stamp and unique ID, that do not contain any information in order to classify a log for a general case. Therefore, a feature extraction where only the most general

and essential tokens are kept.

5.1.2 Feature Extraction

In the feature extraction we remove any form of digits, special characters, and memory addresses. The logs are then transformed to only contain lowercase characters as this improves the robustness and generality. Thus, the original *CoreMW* log is transformed into the following processed log:

```
pl osafamfwd rebooting opensaf nodeid name
no mapped reason amf unexpectedly crashed
ownnodeid supervisiontime
```

Figure 5.2. Example of a processed log from Figure 5.1.

To be able to train and later test the HTM network, the fault label is appended to the end of each log, thus creating the modified processed log on the following form:

```
pl osafamfwd rebooting opensaf nodeid name
no mapped reason amf unexpectedly crashed
ownnodeid supervisiontime coremw
```

Figure 5.3. Example of a labeled system log from Figure 5.2, with the fault label appended in the end of the log.

After this step, the labeled log is appended in a corpus containing all the logs of the data set. This step is necessary as we have to create a word vector space of the words in the logs with the GloVe algorithm. As we need to transform each word to a numerical vector. A pretrained vector space model is not possible for this use case, as the logs contain acronyms that only exist within the data set. Each labeled log is stored for future use in training and testing.

5.1.3 Words to Vectors

To transform each word into a unique vector representation, the GloVe algorithm is trained on the corpus of all log entries. The GloVe algorithm creates a word vector representation of each word. The model parameter of the GloVe algorithms are shown in Table 5.2.

Table 5.2. GloVe parameters used to train the word vector space model. The parameters with bold font indicate use case specific values.

Parameter	Value
VOCAB_MIN_COUNT	5
VECTOR_SIZE	**256**
MAX_ITER	**30**
WINDOW_SIZE	**10**
BINARY	2
NUM_THREADS	8
X_MAX	10

The parameters in bold font in Table 5.2 indicate values specified for the current use case, other parameters kept their original value. The *VOCAB_MIN_COUNT* represents the minimum number of words that are contained in a corpus. The vector size was increased so that the word vector dimension would be greater. This parameter represents the value of the dimension n for word vectors described in subsection 4.1.2. The dimension was increased as the SDR has to be sparse and preferably of a larger dimension, e.g. 1024 or 2048 [16]. However, as Wang et al. [15] showed promising results with one-shot text generation with an HTM network using vector dimension of 50 and 200, a suitable choice of a vector dimension of 256 was selected. To reduce the encoding error of the GloVe algorithm *MAX_ITER* was increased to a larger value. This parameter set the number of iteration that the adaptive gradient algorithm is executed in order to minimise the cost function. The window size is the size of the co-occurrence matrix $\boldsymbol{X}$ described in subsection 4.1.1. It was adjusted to match the average length of the system logs, i.e. 9.5. The possible number of CPU threads used by the GloVe algorithm during training is given by *NUM_THREADS*. Finally, we have *X_MAX* which is the parameter x_{max} defined in Equation 4.4, that

acts as a cut-off where the weighting function is assigned the value of 1.

5.1.4 Semantic Sparse Distributed Representation

For an HTM network to be able to learn the patterns and temporal changes of the SDRs, each bit has to have a unique meaning. This means that the semantic meaning of each word has to be represented by some active bits. The semantic meaning is encoded by the GloVe algorithm, which transforms each word into a numerical vector. The binary representation of the numerical vector is given by computing the bit score vector defined in Equation 4.10. With the bit score vector, the SDR encoding is obtained by applying the encoding function, Equation 4.13, on each bit score vector. Thus, we have an SDR matrix, $\boldsymbol{X}$, where the i'th word vector is represented by $\boldsymbol{X}_i$.

Using experimental analysis it was determined that a sparseness of 2% was deemed too sparse for the temporal memory to learn the input patterns. An increase to 8% was made and satisfactory behaviour of the temporal memory was observed. The parameters used for the encoding of the word SDRs are presented in Table 5.3

Table 5.3. Word SDR parameters.

Length	ON bits	Sparseness
256	21	8.2%

5.2 HTM Network

The HTM network consists of an SDR encoder, HTM module, and an SDR decoder to decode the predictions made by the HTM. The architecture of the Temporal Memory is explained in detail below.

5.2.1 Temporal Memory Architecture

The HTM network used in the task of predicting the fault class of the system logs consists of a single Temporal Memory layer. The architecture of the whole network and the selected parameters are presented in Table 5.4, where

the most important parameters are the *columnCount* and *cellsPerColumn*, together with the synaptic parameter of *activationThreshold*.

The first two parameters specify the number of columns and the number of cells that inhabits each column. The number of columns has to match the dimension of the presynaptic input, thus the value of 256 had to be selected. The number of cells per column determines how many unique patterns the memory can learn. If there are too few, the memory will not be able to learn enough pattern. The default value is 32, and with experimental analysis with the number of cells of $10, 42$, and 56 it was determined that 10 was too few, and values above 32 had no significant improvement. The number of active synapses was decreased from the default value of 12 to 8. This allowed the temporal memory to become more sensitive and able to better detect input patterns.

Table 5.4. Parameters of the Temporal Memory.

Parameter	Description	Value
columnCount	Number of columns in the temporal memory.	256
cellsPerColumn	Number of cells in each column.	32
initialPermanence	Inital permanence value of new synapses.	0.5
connectedPerm	The threshold permanence value for a synapse to become connected.	0.5
minThreshold	If the number of active potential synapses is above this threshold, the dendritic branch has detected a pattern and is eligible for learning.	15
newSynapseCount	The maximum number of synapses that are added to a branch during learning.	12
permanenceInc	The permanence incremental value rewarded to synapses with both active presynaptic and postsynaptic cells.	0.1
permanenceDec	The value of synaptic connections get punished with if the postsynaptic cell is active but presynaptic cell is not, and vice versa.	0.1
activationThreshold	For a dendritic branch to become active at least this number of synapses needs to be active.	8
globalDecay	Decays the permanence value of synapses when it is runs. It will also remove inactive synapses with permanence value of 0 and dendritic branches with no synapses.	0
burnIn	BurnIn evaluates the prediction score of the temporal memory.	1
checkSynapseConsistency	Will preform an invariance check of the synaptic connections if True (1).	0
pamLength	The number of temporal steps that the memory will remain in "Pay Attention Mode" after the end of a sequence. With PAM mode sequences that share elements can be learned faster.	10
verbosity	Controls the diagnostic output.	0

5.3 Classification Algorithm

To be able to classify individual system logs, each labeled log is converted into a sequence. A system log i is represented as the sequence S_i, where $S_i = [w_1, w_2, \ldots, w_n]$ and w_j is the j'th word in that sequence. At time t, the t'th word in the sequence, $S_i^{(t)} = w_t$, is converted to a semantic SDR via the SDR encoder. The encoder converts w_t to the semantic SDR by using

a look up of w_t in the SDR matrix, $\boldsymbol{X}$, and thus outputs $\boldsymbol{X}_t$. The output is then fed to the temporal memory.

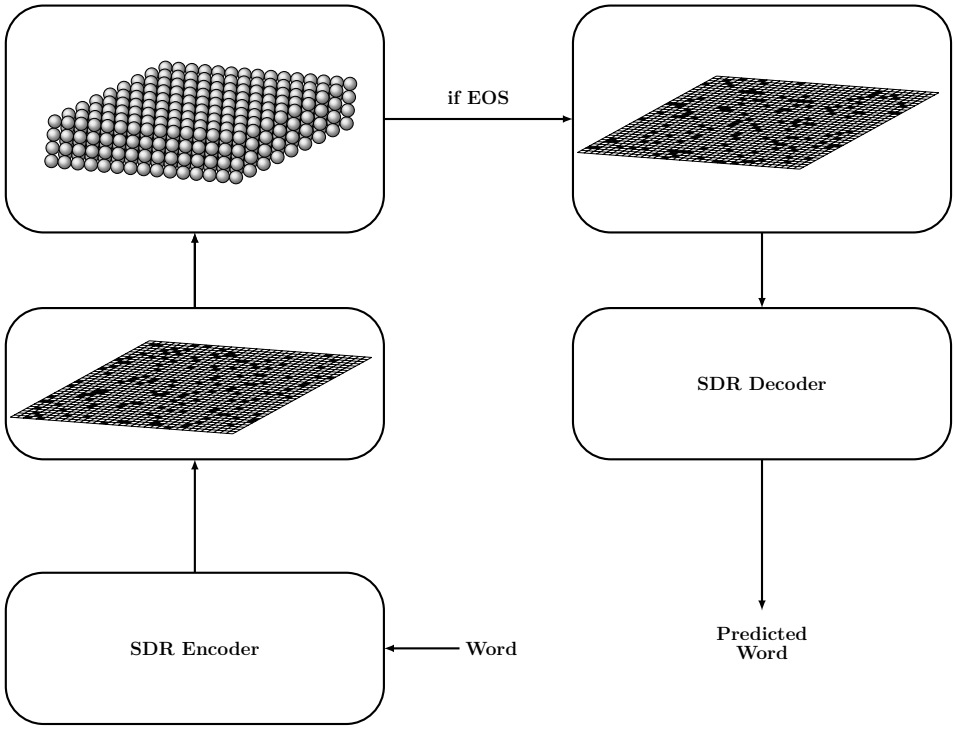

Figure 5.4. An illustration of the architecture of the classification algorithm. A word is encoded via the SDR encoder, which outputs an semantic SDR, illustrated as a gird plane, where black square represents active cells. This SDR is fed to the temporal memory layer, depicted as a stacked layers of spheres, where each sphere is an HTM neuron. If an *End Of Sequence* (EOS) is reached in testing mode, the predicted SDR is extracted and decoded.

5.3.1 Training

During training, the temporal memory uses Hebbian like learning rules to learn input sequences by forming and strengthen synaptic connections, and depressing unimportant ones. The temporal memory is only presented each sequence ones, i.e. trained using *one-shot learning*. As the fault label is at the end of each sequence, represented by w_n, the temporal memory will be able to learn the entire sequence together with the fault label at the end.

Under the training phase no decoding of predicted input SDRs is performed, which is one of the major differences in the training and testing algorithms.

5.3.2 Testing

To test the temporal memory, the original logs are used, thus i'th sequence is represented $S'_i = [w_1, w_2, \ldots, w_{n-1}]$. In the same way as during the training, each semantic SDR in the sequence is fed to the temporal memory. However, at the *End Of Sequence* (EOS), as shown in Figure 5.4, the predicted next presynaptic input, or semantic SDR, is extracted from the temporal memory. The predicted SDR is decoded via the SDR decoder using Equation 4.14. The decoded predicted fault class of the sequence is then logged together with the actual fault label.

5.4 Metrics for Evaluation

To evaluate the HTM model we will use three different type of metrics, *Confusion Matrix*, *Recall*, and *Accuracy*. To understand these metrics we will first introduce the following notations; *True Positives*, *False Positives*, *True Negatives*, and *False Negatives*.

- *True Positives* (tp) – the population of sequences belonging to class i that were correctly classified to be members of class i.
- *False Positives* (fp) – the population of sequences that are not members of class i, but were classified to belong to class i.
- *True Negatives* (tn) – the population of sequences that did not belong to class i and were not classified to be members.
- *False Negatives* (fn) – the population of sequences that belonged to class i, but were classified to belong to some other class.

The first metric we will use for the evaluation of the HTM model is the confusion matrix, as this matrix allows us to visualise the models classification performance. In Table 5.5 an illustration of the general layout of the confusion matrix is presented.

Table 5.5. Illustration of a confusion metric, where *tp* indicates correctly classified examples, and – indicates incorrectly classified examples, if present.

		Predicted Class				
		A	B	C	D	E
Actual Class	A	*tp*	–	–	–	–
	B	–	*tp*	–	–	–
	C	–	–	*tp*	–	–
	D	–	–	–	*tp*	–
	E	–	–	–	–	*tp*

The second metric we will use in the evaluation is called *Recall*. Recall is a sensitivity metric that tells us how good a machine learning model is at correctly classifying the population of examples in class i.

$$Recall_i = \frac{tp_i}{tp_i + fp_i} \tag{5.1}$$

Finally, we will evaluate the accuracy of the HTM agents classifications. Accuracy is a measure that tells us how well the algorithm is at classifying the system log examples correctly. The accuracy for class i is defined as:

$$Accuracy_i = \frac{tp_i + tn_i}{n} \tag{5.2}$$

where tp_i and tn are the population of *True Positives* and is the *True Negatives* for class i. While n is the total number of examples in the data set. However, accuracy the can be misleading if the data set is skewed, as it is in this case. For classes with few examples, the number of tn will have a huge impact on the overall accuracy, as it will always be large. Accuracy is recorded to be able to compare the inference results with the Linnaeus model.

Chapter 6

Results

To evaluate the *Hierarchical Temporal Memory* (HTM) model, described in the previous chapter, the HTM model is trained on the entire data set once, i.e. *one-shot learning*. This was mainly because the data set was not split into a training and test set. The reason for this was due to the skewed distribution of the data set, see Table 5.1, and the lack of shared vocabulary of system logs belonging to the same fault class.

The evaluation of the HTM model is divided into two to parts. First, we measure the performance of the HTM models predictions. Then, we compare them to the statistics collected from the Linnaeus model.

The predictions from the HTM model was obtained by doing inference on seen data, i.e. the entire data set. Three different inference experiments where carried out, where the first test was done with the system logs unaltered. The following two experiments inference was done on system logs with introduced errors. In the second test, ten per cent of the words in each system log were randomly dropped. And in the third test twenty per cent of all words in a sequence were dropped. We call these randomly introduced errors as *"drop-out"* for the remainder of the thesis. These experiments aim to evaluate the robustness of the HTM model to changes in the system logs. The drop-outs represent what would happen if a developer rewrote a system log and changed some of the vocabulary, where some words are not in the GloVe matrix and can therefore not be encoded as SDRs.

6.1 Evaluation of the Inference

The evaluation will be presented in the following order; first, we present the confusion matrix to get an overview of the HTM models performance. Next, the recall of each fault class, to visualise the ability of the HTM model to find fault sequence of a specific fault class. Finally, we have the classification accuracy of each fault class. This gives us an indication of how well the HTM model is at classifying sequences correctly, and at the same time not classifying a sequence to a class it does not belong to.

6.1.1 Inference on Unalterd System Logs

In the first experiment inference was done on unaltered system logs. In Figure 6.1, a comparison between the actual fault categories and the predictions made by the HTM model is visualised. A strong diagonal indicates that most predictions were *True Positives*, i.e. the actual fault label. The *unknown* class indicates sequences when the HTM model could not predict a fault label at all. From the confusion matrix, a observation can be made that the most of the misclassification belong to the *nofault* class. There are two reasons why this type of misclassification happens, first, most of the system logs belongs to this class, see Table 5.1. The second being that the the structure and the words occurring in the system logs are not different enough for the HTM model to distinguish them properly. In Figure 6.2, the recall of each class is presented, with *com*, *lm*, and *sec* being most difficult to correctly classify. An important observation is that these classes has the fewest examples. The accuracy, see Figure 6.3, of the HTM model is high, but as stated before, this measure is saturated by the sheer number of *nofaults*.

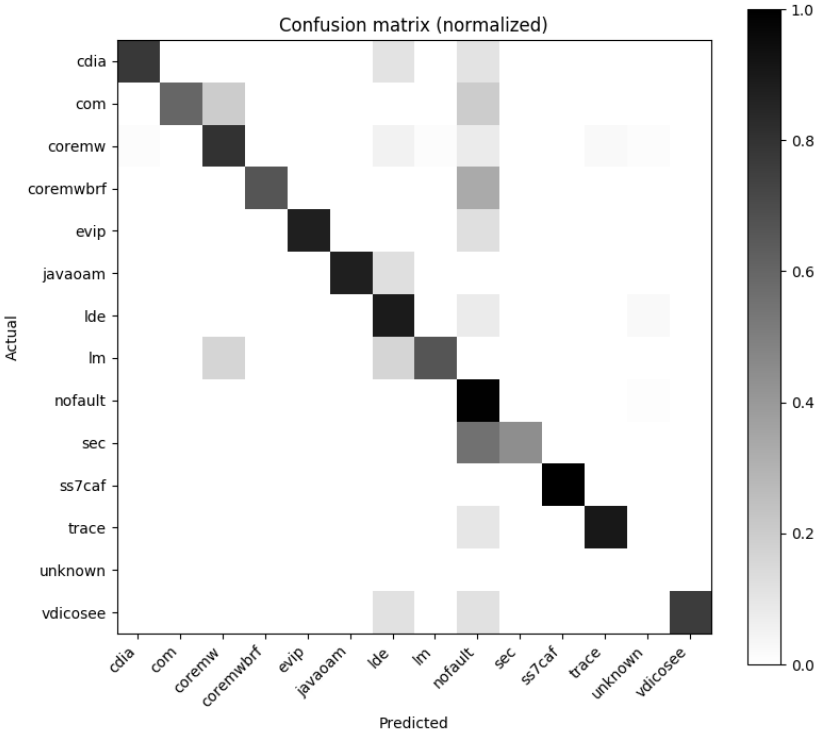

Figure 6.1. The confusion matrix of the classification made by the HTM model with complete system logs. The vertical axis represent the actual sequence presented to the HTM model, and the horizontal axis represent the predicted fault class.

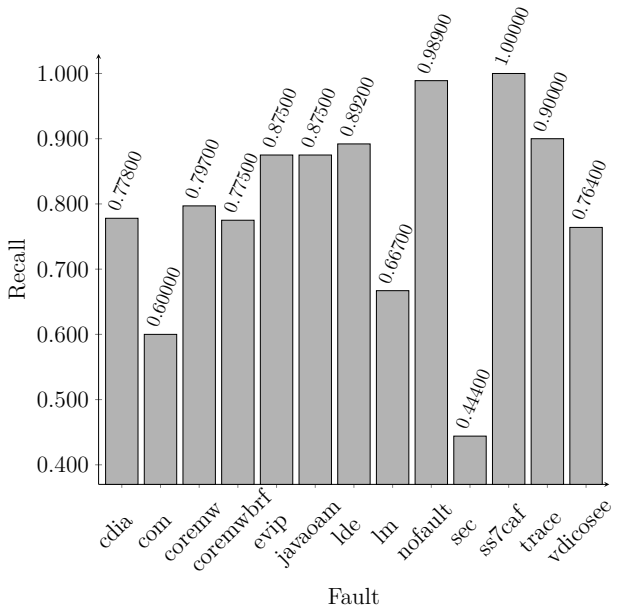

Figure 6.2. Recall of each fault class with complete system logs presented to the HTM model.

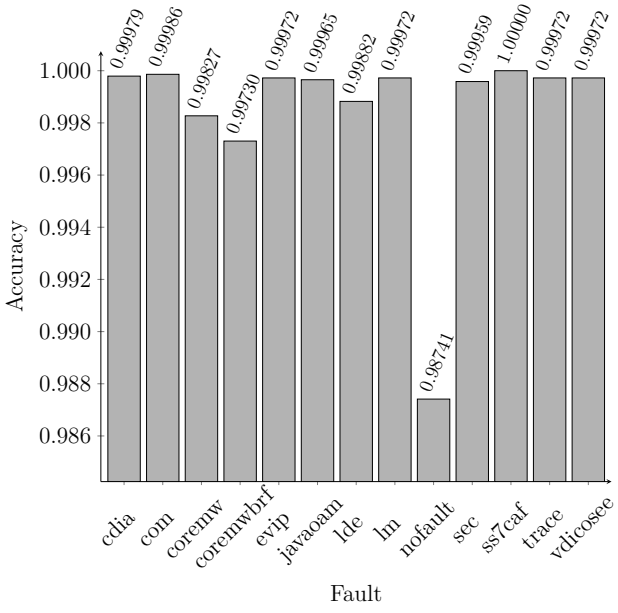

Figure 6.3. Classification accuracy of each fault class with complete system logs presented to the HTM model.

6.1.2 Inference With 10 Per Cent Drop-Out

With a drop-out rate of ten per cent a severe decline in the classification performance is visualised in the confusion matrix, Figure 6.4. In Figure 6.5, the ability of the HTM model to correctly classify the classes with few examples decreases to below 50 per cent in almost all cases. But in Figure 6.6, the accuracy is still very high for all classes except the *nofault* class. The reason for this is that for classes with a small population, the large number of *True Negatives* saturate the metric. However, for the *nofault* class, a major decrease is evident.

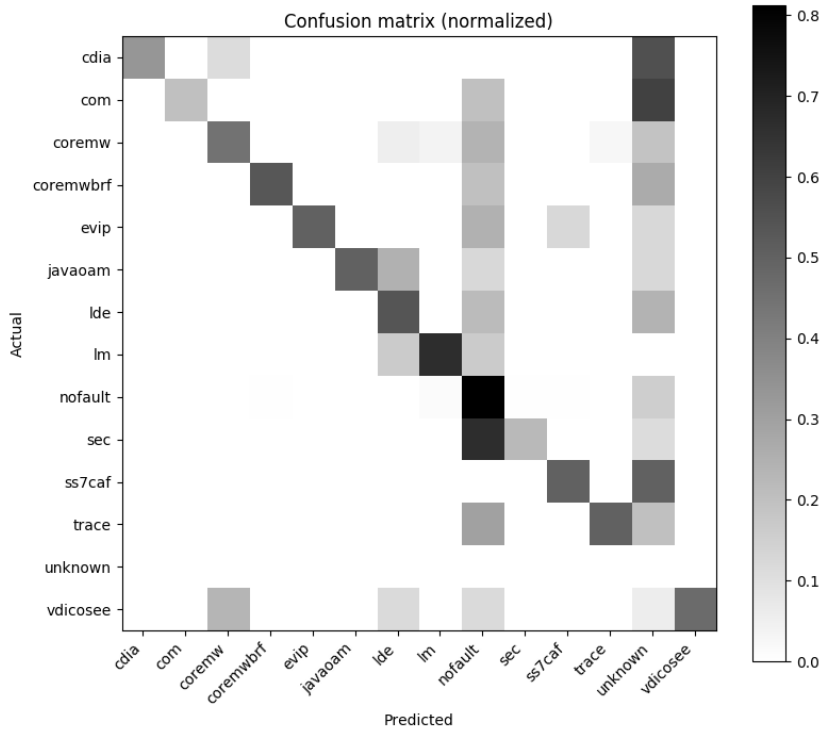

Figure 6.4. The confusion matrix of the classification made by the HTM model with ten per cent word drop-out in each sequence. The vertical axis represent the actual sequence presented to the HTM model, and the horizontal axis represent the predicted fault class.

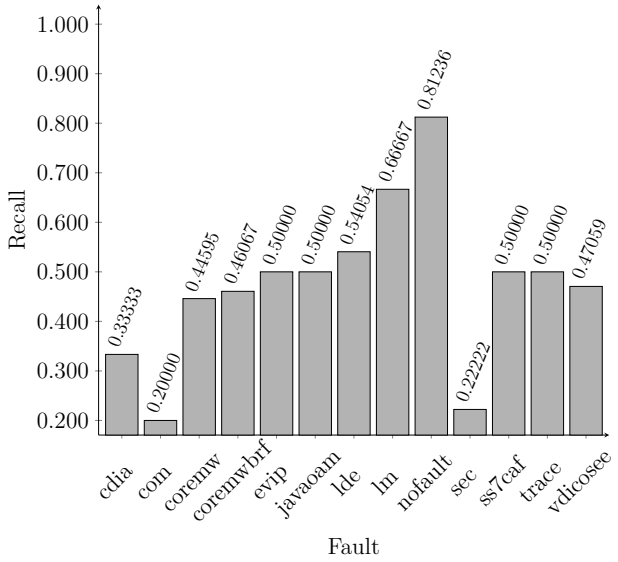

Figure 6.5. Recall of each fault class, with ten per cent word drop-out of each sequence.

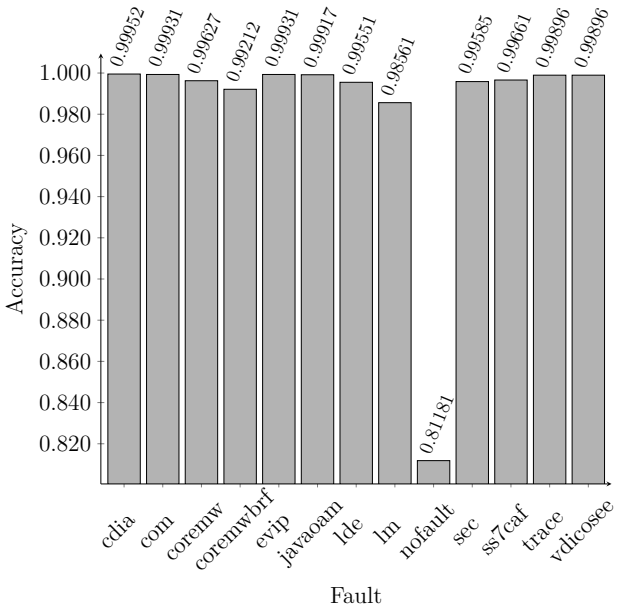

Figure 6.6. Classification accuracy of each fault class, with ten per cent word drop-out of each sequence.

6.1.3 Inference With 20 Per Cent Drop-Out

The trend from the previous experiment is continuing with a drop-out of twenty per cent. The pattern in the confusion matrix, Figure 6.7, is starting to appear random, which means the HTM model has trouble labelling any class correctly in the majority of cases. However, the fault class *ss7caf* is correctly classified in all cases, but from Table 5.1 we can see that *ss7caf* only contain two examples. Thus, no real conclusions can be drawn from this, especially since Figure 6.5 shows that it misclassified *ss7caf* 50 per cent of the time. The severity of the decline is also evident in Figure 6.8, with HTM model having a recall of around 20-30 per cent. In Figure 6.9, the accuracy is still saturated for all classes except *nofault*, which has now dropped to 59.167 per cent.

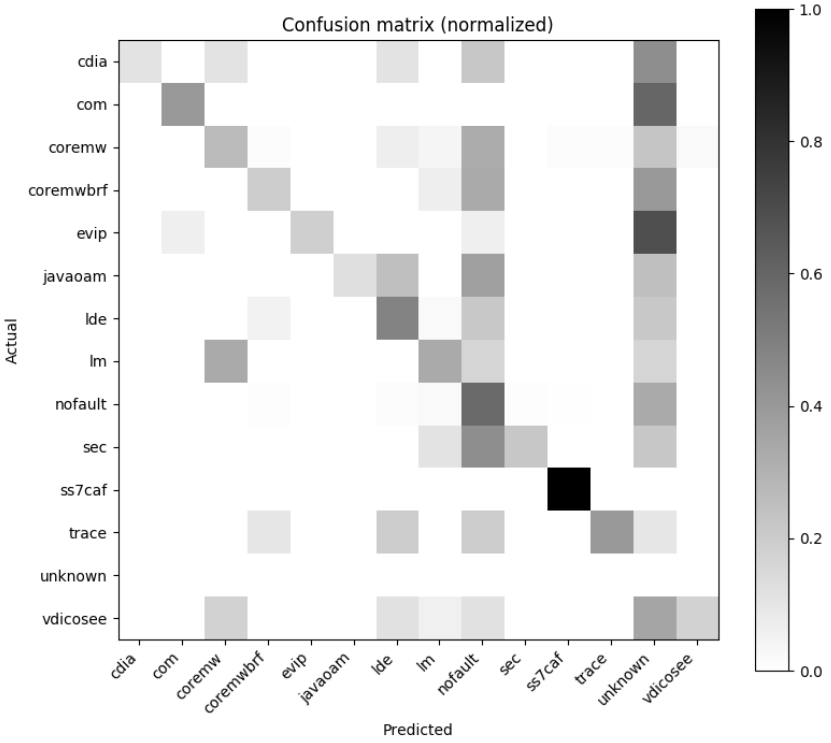

Figure 6.7. The confusion matrix of the classification with twenty per cent word drop-out. The vertical axis represent the actual sequence label presented, and the horizontal axis represent the predicted fault label.

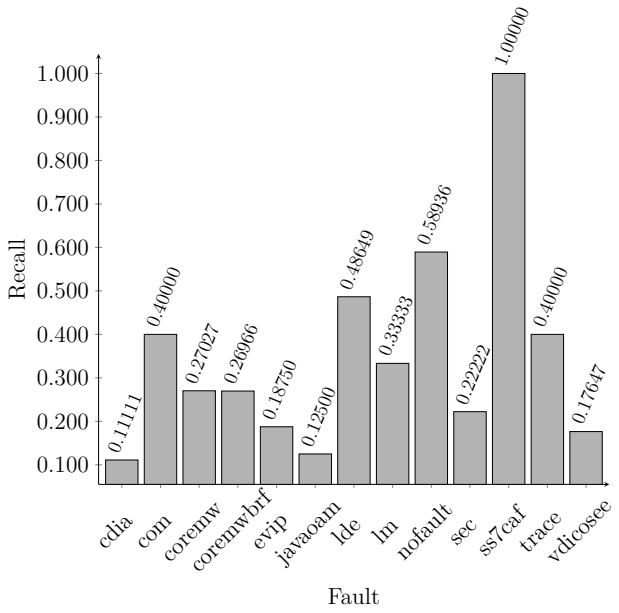

Figure 6.8. Recall of each fault class, with twenty per cent word drop-out of each sequence.

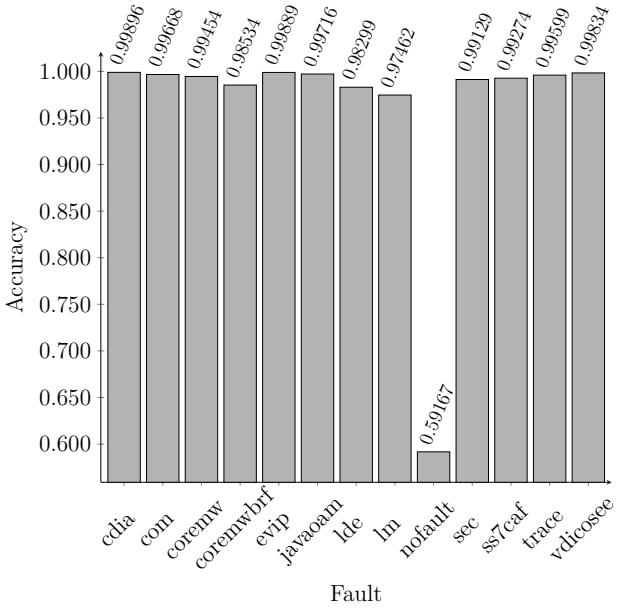

Figure 6.9. Classification accuracy of each fault class, with twenty per cent word drop-out of each sequence.

6.1.4 Comparison With Linnaeus

The final evaluation is a comparison with the already existing classifier *Linnaeus*. If the HTM model should be an viable option in a production setting, it should have comparable or better statistics than *Linnaeus*. However, the machine learning models where benchmarked on different computers, thus contributing to the differences in the timing reports.

The *Linnaeus* model was benchmarked on a computer with an Intel(R) Xenon(R) CPU ES-2658 v2 processor with 40 cores, Skylake microarchitecture, and 2.4 GHz clock frequency. During training 20 cores are utilised at 100 per cent. The HTM model was benchmarked on a computer with an Intel i5-6300U processor with two cores, and with clock frequency of 2.4 GHz. During training and testing of the HTM model one core was utilised at 100 per cent.

In Table 6.1, the comparable statistics are presented. We start by introducing the statistics for the *Linnaeus* model. The first two model types are *word n-grams*, where n contiguous words in each system log are bagged together to form a vocabulary. The next two models are *character n-grams*, and instead of words, n contiguous characters are bagged together. Vocabulary build time represents the time it takes to build the different vocabularies based on the model chosen. The total training time is therefore sum of vocabulary build time and training time. The service side is the occupied memory usage when classifying new system logs. The classification latency represents the time it takes to classify a part of a system log file at a time, currently 88 kB at a time. The test accuracy is based on unseen portion of the data set, i.e. the training and testing set were separated. As shown in Table 6.1, the accuracy for the models is obtain through several runs where training and testing are split different each time. The accuracy range from $70 - 95$ per cent for the different models using the test set, and $98 - 100$ per cent during training.

Table 6.1. The comparable statistics between the two machine learning models.

Linnaeus Model

Configuration	Train Accuracy	Test Accuracy	Vocabulary Build Time	Training Time	Service Size	Classification Latency
Word ngrams = 1	99%	75 − 95%	42 s	150 s	150 MiB	10 ms
Word ngrams = 2	98%	75 − 95%	109 s	217 s	150 MiB	17 ms
Char ngrams = 2	100%	70 − 95%	787 s	787 s	150 MiB	68 ms
Char ngrams = 4	99%	70 − 95%	514 s	514 s	150 MiB	42 ms

HTM Model

Experiment	Train Accuracy	Build Time	Training Time	Service Size	Classification Latency
0% drop-out	98.6%	131 s	828 s	888 MiB	88 ms
10% drop-out	80.7%	–	–	888 MiB	88 ms
20% drop-out	58.5%	–	–	888 MiB	88 ms

Let us now introduce the statistics of the HTM model, also presented in Table 6.1. Instead of model configurations, we present the accuracy obtained from the model during the three experiments performed. We can see a clear degradation of the accuracy as we introduce faults into the system log sequences, from 98.6 to 58.5 per cent. The reason we preformed this test is to see how well the HTM model preformed with added errors to the sequences. This test is however not really comparable to the *Linnaeus* models, but it does tell us how well it is at classifying incomplete system logs. Next, we have the build time, which is the time it takes to perform all the preprocessing steps required. The training time represent the time it took to train the HTM model using *one-shot learning*. Service size is the memory usage of the HTM model. An important thing to note is that the most of the memory usage is due to the allocation lists used for encoding, decoding, and the entire system log data set. The network size is only 178 MiB of the entire 888 MiB. Classification latency is the time it takes to encode each word to an SDR, present it to the HTM model, and decode a prediction at the end of a system log sequence.

From comparing the statistics, the *Linnaeus* models achieves similar results in the training cases as the HTM model. The classification of the

system logs are performed in different ways, where the *Linnaeus* model takes system log chunks and processes them, where as the HTM model looks at individual logs, one at a time. It is likely that the execution time for all parts of the HTM model would be faster if it where evaluated on a better machine. But with current metrics, the *Linnaeus* model has the benefit have lower classification latency and smaller memory footprint, i.e. service size.

The test with 10 and 20 per cent dropout with HTM model is interesting, as they give some indication on how well the model is at being able to classifying system logs if the vocabulary is changed. However, with the current HTM model we can see that it does not adapt well to introduced errors in the system log sequences.

Chapter 7

Discussion and Conclusion

In this chapter we discuss and summarise the results obtained and presented in the previous chapter. We start off with explaining why we used the entire dataset for training and testing. Then, we discuss the results and the evaluation. Next, we discuss the model and the design choices made. Finally, we give a conclusion of this work, future improvements, and ethics.

7.1 Discussion

The main objective of this thesis was to create and compare an *Hierarchical Temporal Memory* (HTM) model with the existing system called *Linnaeus*, in order to evaluate if the HTM algorithm is a suitable choice for anomaly detection of system logs. First, we discuss the results presented in chapter 6. Then we will go over the choice of method where we discuss the data and the HTM model. An important note to make before we start the evaluation, is to state that the HTM model was trained and tested on the entire data set. The choice for this will be discussed further under the *Data* subsection.

7.1.1 Data

The data set contained a set of examples of system logs classified by a technician to be the reason for causing a specific fault. During the thesis there was no possibility of expanding this data set with more examples. To manually go through unlabelled data sets would require domain expertise.

Due to the fact that the HTM model was evaluated on the data set that it was trained on. It is important to state the reason why before we start discussing the performance of the algorithm. The decision not to split the data set into a training and testing set was made for the following reasons:

- The data set was heavily skewed, where 95.2% of 14459 labeled examples belonged to the *NoFault* class.
- For some of the classes with few examples, there was no underlying pattern, meaning that they did not use the same vocabulary. Therefore, no machine learning algorithm would be able to correctly classify the other system log while only being trained on the first example.

7.1.2 Results

The results gathered to evaluated the HTM model are based on three metrics describe in section 5.4. First, we used the *Confusion Matrix* to visualise the performance of the classifications. Then, we presented the *Recall* and *Accuracy* for each experiment.

During the first experiment we did not alter the system logs in any way. The performance of the HTM model is rather impressive even though it is tested on the training set. The reason that the results show promise is that the model was trained using *one-shot learning*. But it is also impressive due to the distribution of the dataset, where there is a possibility that any machine learning model would over-fit on the *NoFault* class.

It would however been interesting to evaluate the system using *multi-shot learning*. But as the entire dataset was used for training and testing, the risk of over-fitting would have been great. If the distribution of the samples in the dataset was different, *multi-shot learning* would have been an interesting choice, where the model would be evaluated on unseen examples.

However, with *one-shot learning* the HTM model it is less likely to over fit on during training. Therefore, some conclusions can be drawn from the results gathered. As we see in Figure 6.1, the diagonal is prominent for the first experiment, meaning that the model is able to make correct predictions most of the time. If we then move on to study the recall and accuracy figures, shown in Figure 6.2 and Figure 6.3 respectively, we see that the model had an overall satisfactory performance, meaning that it had over 70% recall on most classes. The accuracy throughout all experiments does not tell us a whole lot about the performance of the model. The reason for

this is that this metric is saturated by the sheer size of *True Negatives* for all classes except the *NoFault* class.

The trend of decay in performance is present in both of the following experiments when errors in the sequences are introduced. Which is a likely scenario if new words are introduced by a developer. A reason for this poor performance has to due with how the HTM algorithm learns sequences. Much like the a human with reading comprehension can recite the alphabet forward, every thing is done in sequence of sequences, e.g. *"abcd"* then *"efg"*. But if a human asked to change something in this learnt sequence before reciting it, e.g. reciting the alphabet backwards, he will find it impossible to do, unless he knows it beforehand. This is because the brain and the HTM algorithm learns forward sequence of sequences [9], [17]. When a sequence with drop-out is presented to the HTM algorithm, the columns will be *bursting* due to the unpredicted input pattern, which will have consequences on future predictions.

Evaluation

The evaluation is not obvious to make based on the acquired results. This is mainly due to the fact that the accuracy is not a good metric on a data set with an uneven distribution between the classes. This is due the fact that the size of the *True Negative* population will saturate the accuracy for the classes with few examples. To quantify the performance of the two models, access to real test logs files would almost be necessary. But to do a fair comparison between the models with the statistics presented in Table 6.1, we should use the training accuracy of the *Linnaeus* model. We can then see that they are almost identical in performance. However, the results from the *Linnaeus* model could be over-fitted, due to the fact that most no noise or errors, e.g. drop-out, where introduced during training or testing. The advantage of HTM model is that it used *one-shot learning*, thus minimising the risk of over-fitting. However, HTM model does not optimise over a parameter space either, as it uses a Hebbian style learning, which should make it less prone to over-fitting the data in any case.

In terms of classification speed, *Linnaeus* is between 11 and 48 per cent faster. The computational time could largely be due to the fact that they where trained and tested on two different computers, *Linnaeus* on a powerful desktop and the HTM model on a laptop. However, the NuPiC framework that the HTM model uses for the temporal memory algorithm does not

support multi-core processing. So the speed-up would be somewhat limited. With this said, the HTM model could be optimised by looking over parts of the code that is inefficient, e.g. the encoding and decoding function.

An obvious benefit of using a system log classifier based on the HTM algorithm is that when new system logs are introduced, retraining on the whole system log database is not needed. It would be enough to only present the new system logs to the HTM model. This is because the HTM model does not optimise weights given a certain input, as in traditional machine learning techniques. Instead, input patterns are learned via the change of permanence values in the synaptic connections of each HTM neuron. Thus, making the model robust to changes in the system logs.

7.1.3 Method

In this subsection we will discuss the design choices made in the implementation of the model.

Model

There are two parts in the implementation that could be analysed; the encoding of words into Sparse Distributed Representations (SDRs), and secondly the network architecture.

The performance of the HTM model is heavily dependent on how well information is encoded into a Sparse Distributed Representations (SDRs). A drawback to our implementation is that it is hard to evaluate the GloVe encodings, as the word vector space is created by a corpus of all system logs. The system logs themselves, as shown in Figure 5.2, are not structured messages in the sense that analogies that be derived from them. This could in turn mean that the GloVe encodings are sub optimal when encoding words into SDRs. This could mean that dissimilar words or acronyms are encoded as SDRs that share on bits, which could then confuse the temporal memory to confuse system log sequences.

Lastly, the HTM model is a one layer temporal memory with 32 cells in each of the 256 columns. The reason for limiting the network to one layer was that predictions of future inputs from this layer could be easily extracted and decoded. Reports, such as [15], indicates that one layered temporal memory is enough to create a state-of-the-art text generation algorithm. With more

layers the size of the service size of the model would have increased, which was not inline with goal of this thesis.

7.2 Conclusion

The HTM model has shown promise in classification of system logs. It preforms well with limited computational resources and is able to classify system logs rather well with *one-shot learning*. The author believes that a claim regarding the use of an HTM model in a production setting would be easier if a more thorough benchmark was made, e.g. system log files from real test was presented to both classifiers, and then compare metrics. But with the current knowledge, the author believes that an HTM model for this task would be preferable due to the fact that the model does not need to be retrained on the entire data set when new examples are introduced.

7.3 Future Work

The first major improvement to be made would be to train the HTM model on system log sequences with different drop-out rates to see if the HTM model is able to learn to generalise.

The drop-out experiments tries to recreate the situation when a new word is introduced and the encoder is not able to encode it into an existing SDR. However, instead of just passing on to the next word in the sequence, it would be interesting to pass the prediction from the previous time step in as the input SDR to the network. This would prevent some columns from bursting, where all cells become active, and minimise this impact on the future prediction and the prediction of the fault class.

The next improvement could be to change the structure of the HTM model and remove the temporal aspect of viewing the system logs, and instead use an approach similar to *coritcal.io* [18] with a semantic fingerprinting (SDR) of each system log. The semantic fingerprints would be created by, as before, first creating a unique SDR for each word. By aggregating all the SDRs of a sequence together, each index would increment for each ON bit in each SDR. To preserve the spareness of the new SDR, a threshold would be introduced to only convert the indices with the largest accumulated values into a new sparse SDR, while the indices below the threshold would be converted to OFF bits. These semantic fingerprints would then

be presented to a spatial pooler, which is a temporal memory without the temporal aspect taken into account. The advantage of this approach would be that the noise tolerance of the HTM theory could be used to its fullest potential, as noise of up till 50 per cent in an input SDR can be handled by an spatial pooler or temporal memory [9].

Another interesting approach would be to find patterns of conditional probabilities between logged events, in order to find faults in system log files. This task could be suitable for an HTM algorithm due to the sequential nature of this task.

A final interesting improvement would be to translate all the Ericsson acronyms into English words. This would then mean that a GloVe model would not be needed to be retrained when new ones appear. We might then be able to use a pretrained GloVe model on the first 6 billion tokes on English Wikipedia.

7.4 Ethics

The ethical aspects of this thesis and the proposed implementation does not in its current form present much of an ethical issue. This is largely because this model is contained to only analyse system logs gathered in testing. Therefore, the impact of its use is limited, as it will not impact a product providing a service to Ericsson's customers. The intent of this system is not to replace anyone, as this will only be a tool in which a technician could rely on when doing troubleshoots and classifying system log files correctly. And if the system did not perform well enough to reliably help a technician, the system would not be used.

The ethical aspect could instead be discussed on the topic of machine learning at large. Where there are numerous ethical issues that could arise when implementing machine learning software. From use in autonomous cars to weapon systems. But as these subjects are not part of this thesis we will not dwell further into these never ending philosophical debates.

Bibliography

[1] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic Routing Between Capsules", *CoRR*, vol. abs/1710.09829, 2017. arXiv: 1710.09829. [Online]. Available: http://arxiv.org/abs/1710.09829.

[2] K. J. Han, A. Chandrashekaran, J. Kim, and I. R. Lane, "The CAPIO 2017 Conversational Speech Recognition System", *CoRR*, vol. abs/1801.00059, 2018. arXiv: 1801.00059. [Online]. Available: http://arxiv.org/abs/1801.00059.

[3] Y. Liu and M. Lapata, "Learning Structured Text Representations", *CoRR*, vol. abs/1705.09207, 2017. arXiv: 1705.09207. [Online]. Available: http://arxiv.org/abs/1705.09207.

[4] A. Catovic. (Oct. 2017). Linnaeus - Fault Detection and Classification Service.

[5] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin, "Biological and Machine Intelligence (BAMI)", Initial online release 0.4, 2016, [Online]. Available: http://www.numenta.com/biological-and-machine-intelligence/.

[6] E. Drake, I. El Khayat, R. Quinet, E. Wennmyr, and J. Wu, "Paving The Way To Telco-Grade PAAS", *Charting The Future Of Innovation Ericsson Technology Review*, vol. 93, no. 4, May 2016. [Online]. Available: https://www.ericsson.com/assets/local/publications/ericsson-technology-review/docs/2016/etr-telco-grade-paas.pdf.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0201745720.

[8] J. Hawkins and S. Blakeslee, *On Intelligence*. New York, NY, USA: Times Books, 2004, ISBN: 0805074562.

[9] J. Hawkins and S. Ahmad, "Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex", *Frontiers in Neural Circuits*, vol. 10, p. 23, 2016, ISSN: 1662-5110. DOI: 10.3389/fncir.2016.00023. [Online]. Available: https://www.frontiersin.org/article/10.3389/fncir.2016.00023.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning.(Report)", eng, *Nature*, vol. 521, no. 7553, May 2015, ISSN: 0028-0836.

[11] J. Hawkins, S. Ahmad, and Y. Cui, "A Theory of How Columns in the Neocortex Enable Learning the Structure of the World", *Frontiers in Neural Circuits*, vol. 11, p. 81, 2017, ISSN: 1662-5110. DOI: 10.3389/fncir.2017.00081. [Online]. Available: https://www.frontiersin.org/article/10.3389/fncir.2017.00081.

[12] S. Ahmad and J. Hawkins, "How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites", *CoRR*, vol. abs/1601.00720, 2016. arXiv: 1601.00720. [Online]. Available: http://arxiv.org/abs/1601.00720.

[13] ——, "Properties of Sparse Distributed Representations and their Application to Hierarchical Temporal Memory", *CoRR*, vol. abs/1503.07469, 2015. arXiv: 1503.07469. [Online]. Available: http://arxiv.org/abs/1503.07469.

[14] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation", in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162.

[15] Y. Wang, Y. Zeng, and B. Xu, "SHTM: A neocortex-inspired algorithm for one-shot text generation", in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2016, pp. 000 898–000 903. DOI: 10.1109/SMC.2016.7844355.

[16] S. Purdy, "Encoding Data for HTM Systems", *CoRR*, vol. abs/1602.05925, 2016. arXiv: 1602.05925. [Online]. Available: http://arxiv.org/abs/1602.05925.

[17] R. Kurzweil, *How to Create a Mind: The Secret of Human Thought Revealed*. New York, NY, USA: Penguin Books, 2013, ISBN: 9780143124047.

[18] F. D. S. Webber, "Semantic Folding Theory And its Application in Semantic Fingerprinting", *CoRR*, vol. abs/1511.08855, 2015. arXiv: 1511.08855. [Online]. Available: http://arxiv.org/abs/1511.08855.

[19] J. H. Kaas, "Neocortex", eng, in *Encyclopedia of the Human Brain, Four-Volume Set*, pp. 291–303, ISBN: 978-0-08-054803-6.

[20] S. Lodato and P. Arlotta, "Generating Neuronal Diversity in the Mammalian Cerebral Cortex", eng, *Annual Review of Cell and Developmental Biology*, vol. 31, no. 1, pp. 699–720, Nov. 2015, ISSN: 1081-0706.

[21] H. N. Martin, *The human body: an elementary text-book of anatomy, physiology, and hygiene : including a special account of the action upon the body of alcohol and other stimulants and narcotics*. Holt., 1886. [Online]. Available: http://hdl.handle.net/2027/hvd.hc4bm2.

[22] A. Crossman and D. Neary, *Neuroanatomy E-Book: An Illustrated Colour Text*, ser. Illustrated Colour Text. Elsevier Health Sciences, 2014, ISBN: 9780702 054068.

[23] H. Braak, *Architectonics of the Human Telencephalic Cortex*, eng, ser. Studies of Brain Function, 4. 1980, ISBN: 3-642-81522-7.

[24] M. Marín-Padilla, "The mammalian neocortex new pyramidal neuron: a new conception", *Frontiers in Neuroanatomy*, vol. 7, p. 51, 2014, ISSN: 1662-5129. DOI: 10.3389/fnana.2013.00051. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnana.2013.00051.

[25] V. Mountcastle, "The columnar organization of the neocortex", eng, *Brain*, vol. 120, no. 4, pp. 701–701, Apr. 1997, ISSN: 0006-8950. [Online]. Available: http://search.proquest.com/docview/20611728/.

[26] J. h. Lui, D. v. Hansen, and A. r. Kriegstein, "Development and Evolution of the Human Neocortex", eng, *Cell*, vol. 146, no. 1, pp. 18–36, 2011, ISSN: 0092-8674.

[27] G. A. G. (A. G. Mitchell and E. L. Patterson, *Basic anatomy*, English. Edinburgh : E. & S. Livingstone, 1954.

[28] V. B. Mountcastle, "Introduction", *Cerebral Cortex*, vol. 13, no. 1, pp. 2–4, 2003. DOI: 10.1093/cercor/13.1.2. [Online]. Available: http://dx.doi.org/10.1093/cercor/13.1.2.

[29] E. J. Capaldi, "THE ORGANIZATION OF BEHAVIOR", *Journal of Applied Behavior Analysis*, vol. 25, no. 3, pp. 575–577, Sep. 1992, ISSN: 0021-8855.

[30] K. J. Hole, "The HTM Learning Algorithm", in *Anti-fragile ICT Systems*. Cham: Springer International Publishing, 2016, pp. 113–124, ISBN: 978-3-319-30070-2. DOI: 10.1007/978-3-319-30070-2_11. [Online]. Available: https://doi.org/10.1007/978-3-319-30070-2_11.

[31] J. Defelipe and I. Fariñas, "The pyramidal neuron of the cerebral cortex: Morphological and chemical characteristics of the synaptic inputs", eng, *Progress in Neurobiology*, vol. 39, no. 6, pp. 563–607, 1992, ISSN: 0301-0082.

[32] N. Spruston, "Pyramidal neuron", eng, *Scholarpedia*, vol. 4, no. 5, 2009, ISSN: 1941-6016.

Appendix A

The Neuroscience of the Neocortex

This is a complementary chapter to *"A Brain-Inspired Algorithm"*. We will go through some of the neuroscience of the *new cerebral cortex* (neocortex), that the *Hierarchical Temporal Memory* (HTM) theory is biologically derived from. We being by introducing the neocortex of the mammalian brain and why it is believed to be the part of the brain where intelligence resides. Then, the topology of the neocortex is presented, where we will go through the structure of each layer. Next, we describe the columnar organisation of the neocortex, first coined by Vernon B. Mountcastle, which is one of the main inspirations in HTM theory. We then continue to describe the neural circuitry of these cortical columns. Finally, we introduce the pyramidal neuron, which is the most frequently occurring type of neuron that resides in the neocortex. The HTM neuron is modelled on the pyramidal neuron's functionality and the active processing properties of its dendrites.

A.1 The Neocortex

What separates all mammals from the rest of the animal kingdom is the existence of the neocortex in the cerebral cortex [19]. The neocortex is perceived as the part of the brain where intelligence resides. The reason for this is that the neocortex is responsible for higher order brain functions, such as sensory perception, advanced motor control, association, memory, and cognition [20].

The forebrain, shown in figure Figure A.1, is the largest part of the mammalian brain, made up of the cerebrum, thalamus, and hypothalamus. The

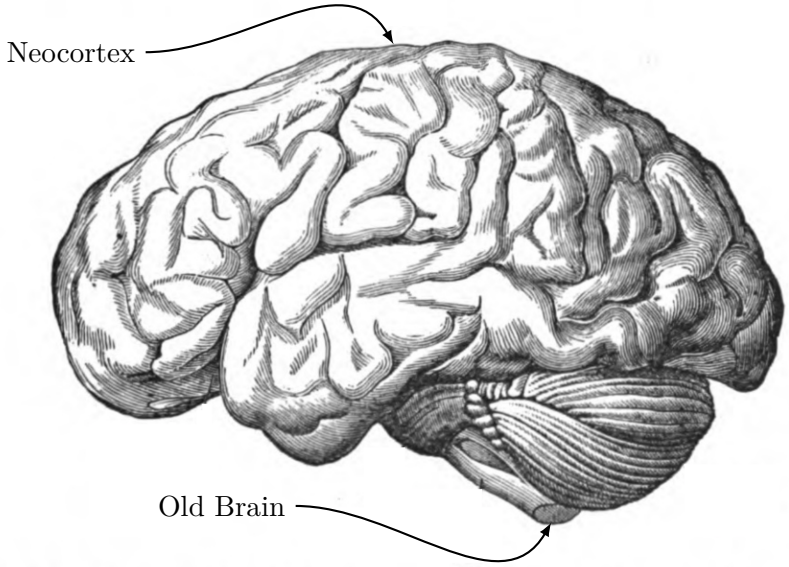

Figure A.1. [21] A sketch of the brain of a Homo sapiens. The cerebral hemisphere is recognised by the convoluted form with ridges and furrows.

cerebrum is made up of two cerebral hemispheres. The cerebral hemisphere has an outer layer of grey matter and the cerebral cortex beneath it [22]. The cerebral cortex consists of two areas, the neocortex and the allocortex, defined by their cytoarchitecture [23]. The neocortex is larger in area and it is the outer most layer of the cerebral cortex, organised into six horizontal laminae of neurons. The allocortex is smaller in size with subareas such as the hippocampus, subiculum and olfactory cortex [19]. Because mammals share the same brain structure of the olfactory cortex and the hippocampus with reptiles but not the neocortex, it was long viewed that the neocortex was a new evolutionary addition of the brain, hence the name. However, the neocortex should instead be viewed as homologous with the dorsal cortex of reptiles. Though, the difference between them is noticeable, the neocortex is a thick, laminated and uniform structure containing abundance of neurons; compared to the dorsal cortex in reptiles which contains only about a row of neurons in its sheet of tissue [19]. With these dissimilarities in mind,

the function of all vertebrate brains is still the same, which is to act as a premotor organ that sends motor commands to the animals' muscles [24].

What makes the human neocortex so special that its able to give us humans the ability to express our thoughts through motor commands, e.g. writing or speaking, perception, imagination, language processing and cognition? One major factor to why humans surpasses it counterparts in intelligence is the sheer size of the neocortex, a surface area of roughly 2600 cm^2 and a thickness of 3-4 mm, together with the vast number of neurons, up to 28×10^9, that resides there [25]. The human neocortex wraps around the forebrain in a convoluted fashion, compared to a rodents' neocortex which is non-folded. This allows the human neocortex to have more surface area with a limited volume [26]. Another reason why humans surpasses all other mammals is evolutionary, i.e. having a large neocortex with many subdivisions is metabolically costly, takes a long time to develop and mature and is unnecessary for most other animals [19].

A.1.1 The Topology

The neocortex organized in hierarchy of six horizontal stacked laminae, numbered I–VI, with each of the layers grouped based on synaptical connectivity, morphological type of neurons and at which depth they exist [19]. This basic structure is shown in Figure A.2. To understand the difference of each layers' topology and processing properties, a brief description is given below:

- **Layer I:** The top layer, which is mostly inhabited by fibers that run parallel to the surface. The fibers consist of the ends of apical dendrites and axons that connect to each other via synapses, together with a few neurons. This layer serves two purposes; first, to work as a feedback connection from other cortical areas. Secondly, to modulate input from the brain stem [19].

- **Layer II:** This layer has a large population of small neurons. The dendrites of these neurons are connected to neighbouring cortical areas and pyramidal neurons in deeper layers via their apical dendrites. Thus, allowing intracortical connections to affect the processing in a local area [19].

- **Layer III:** One of the main output layers of the neocortex, consisting of medium sized pyramidal neurons with basal and apical

dendrites. The axons of these neurons serves as commissural fibers, i.e. connections between the two cortical hemispheres, and as connections to other cortical areas [22]. The pyramidal neurons receive their input form other layers in the local area via their apical dendrites. Adjacent neurons in layer III are able exchange information via their basal dendrites. This layer is of great importance, as it makes it possible for information exchange between regions in the neocortex. Thus increasing the complexity of possible operations that can be performed [19].

- **Layer IV:** The main input layer and situated in the middle occupied by small stellate cells. The name is given due to the radial propagation of the dendrites. The information that the stellate cells receives is shared locally between neighbours via the basal dendrites and then transmitted to the adjoining layers above and below. The stellate cells are small due to the fact that the axons and dendrites only extend a short range [19].

- **Layer V:** This is the other main output layer of the neocortex. Large pyramidal neurons inhabit this layer, which can be seen in Figure A.2 as triangular black spots. The main task of this layer is to integrate information from both its horizontal neighbours and all vertically connected neurons in the layers above up till layer I that belongs to the same cortical column [19]. This information is then sent out to extra-cortical parts of the brain, i.e. basal ganglia, thalamus, brain stem and the spinal cord, via long axons [22]. But also to remote cortical areas, including areas in the other cerebral hemisphere. As these distances can be quite far and information needs to travel fast, the axons needs thick, as this allows for faster conduction rates. To gather information from the vertically adjoining layers, the apical dendrites branch out in long vertical fibers, with synaptic connections in each layer. Neighbouring cells communicate via their basal dendrites. For the pyramidal neurons to be able to send and receive action potentials on their long apical dendrites, and also thick axons the cell body needs to be large [19].

- **Layer VI:** Until this layer, the information has just been processed in a feed-forward manner, i.e. receive information, manipulate it and transmit it onward to other regions. But bi-directional com-

munication is necessary in any complex computational unit, and this is where layer VI comes in. This is the main feedback layer to other activation regions about the state of the local area. The pyramidal neurons that resides in layer VI mainly get their input for layer IV, as well as direct input from axons that terminate in layer IV. With these connections, layer VI is able to project back information to the neurons in another cortical area that contributed to the activation [19].

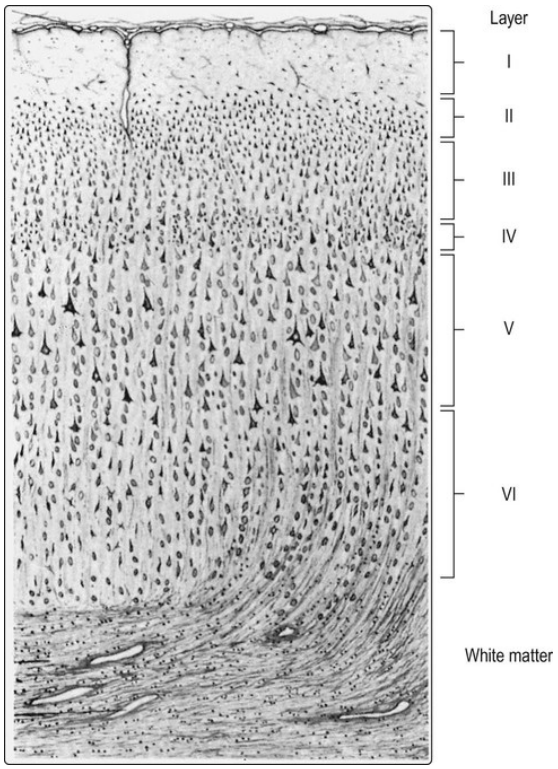

Figure A.2. [27] The cross section above, visualises the general structure of the neocortex categorised into six horizontal laminae, I–VI, on the basis of cytoarchitecture. The cell bodies of the neurons are shown as the darkened spots, neither the axons nor dendrites of the neurons are visible. The white matter connects the neocortical region to other regions in the neocortex along with other parts of the brain. These connections allows the region to send and receive signals [19].

With the understanding of each layers' properties we can now move on to explain the most basic processing unit of the neocortex, the *minicolumn* and the *cortical column*.

A.1.2 Columnar Organisation

The most basic processing unit in the neocortex is called a *minicolumn*, first described by Vernon B. Mountcastle in 1955–1959 [28]. A minicolumn is a vertical array of neurons connected to each other like a chain across layer II–VI. These minicolumns have transverse diameter of 50–80 μm and contain between 80–100 neurons in primates, but can contain up to 2 times more in the striate cortex. The structure of the minicolumn can be observed throughout the neocortex, however they are not identical as some features differ, e.g. cell phenotypes, synaptic transmitters [28]. A *cortical column* is a group of minicolumns which is formed by a common input and the horizontal links they have with each other. The number of minicolumns that a cortical column contains varies between 50–80 [28]. The cortical column functions as a complex distributed system, where each minicolumn can be viewed as a computing node, which is able to modulate inputs and use bilateral communication with neighbouring minicolumns and cortical columns. This allows the cortical columns to link multiple inputs to multiple outputs [25].

A.1.3 Neural Circuit Functionality

In each minicolumn there exists two types of neurons; the *excitatory neurons*, i.e. pyramidal and spiny stellate cells; and the *inhibitory neuron*, the aspiny stellate cells. The neurons are distributed throughout the layers in the minicolumn with the majority, $\sim$80% , of the neurons being of the excitatory type [19]. The excitatory neurons are able to depolarise the neurons of which it is synaptically connected to by sending out a excitatory neurotransmitter, glutamate. In this depolarised state the effected neuron is more likely to itself send out an action potential if it receives strong enough signal on its dendrites [19]. The opposite is true for the inhibitory neuron, as it sends out an inhibitory neurontransmitter, i.e. a biochemical called *gamma-Aminobutyric acid*, or GABA, which hyper-polarises the effected neuron, making it less likely to produce an action potential. This inhibitory effect allows a local circuit to remain sparse, as minicolumns that respond to the

same input tend to inhibit each other. This limits the response time for an input and makes the cortical column sensitive to temporal changes [19].

The cortical columns are able to correlate different inputs based on interconnections that exist in layer III, as these makes horizontal connection in between cortical columns. The advantage of such connections are that neurons in different cortical columns can become active at the same time, thus being able to respond to the signal of the same object and giving it cohesion [19]. Another crucial part of the neural circuits is its plasticity. The synaptic connections between neurons change over time, either they become stronger or weaker, which makes the brain able to adapt to new input. This is largely due to the two types of membrane receptors that exist, i.e. the *N-methyl-D-aspartate*, or NMDA, receptor and the non-NMDA receptor. The non-NMDA receptor reacts when glutamate is released, which leads to depolarisation of a neuron. When this happens, the NMDA receptor, which is otherwise blocked by magnesium ions, is unblocked and can pass calcium ions through if another glutamate release occurs in a short amount of time. This leads to a fast succession of action potential being generated by the neuron. Because of this, synaptic strength between the the neurons is increased, which is called long-term potentiation. The opposite is called long-term depression, which means that the neurons does not respond to the same input and their synaptic connections are weakened [19]. This alteration of the synaptic strength between neurons is also known as Hebbian learning [29].

It is in these synaptic connections that memories are stored. Memories are stored in a sequences and can be recalled auto-associatively. This makes the neocortex robust to noise and incomplete input patterns, as it can autocomplete as input patters as memories are recalled in an ordered sequence. Thus, the neocortex is able to make predictions about the future based on incomplete input [30].

A.2 Pyramidal Neuron

The most abundant of all neuron types in the neocortex is the pyramidal neuron. The pyramidal neuron is unique to the cerebral cortex and can be found in all layers except the first and represent between 70-85% of the total population in any cortical area. It has therefore been seen as the principle neuron of the cerebral cortex [31]. As the name indicates, the cell body, or

soma, of the pyramidal neuron has a pyramidal shape, which can be seen in Figure A.3.

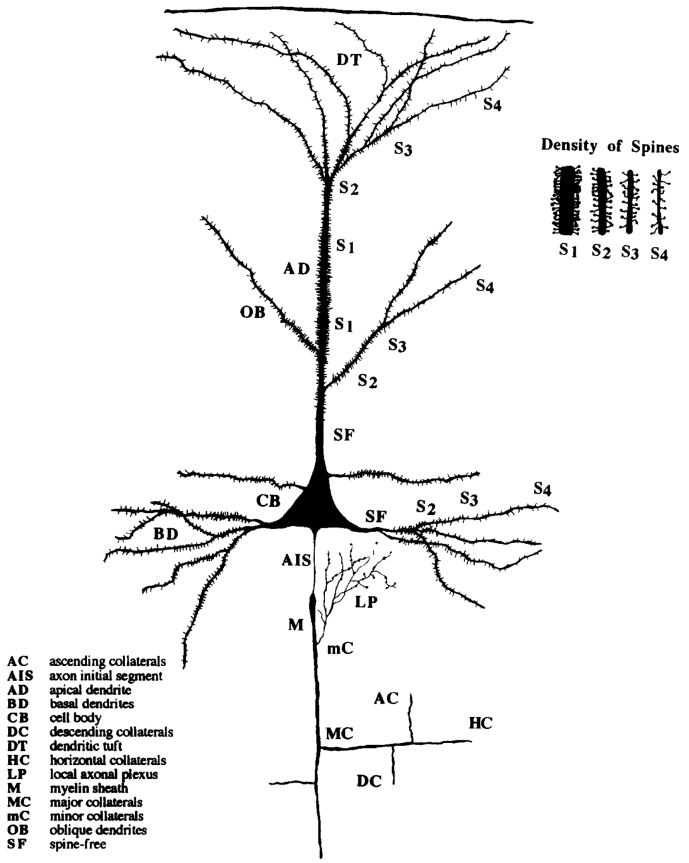

Figure A.3. [31] A sketch of a stereotypical pyramidal neuron, with a apical dendrite branching out from the apex of the soma, the basal dendrites extends out in the horizontal direction and the axon is descending from the soma.

A.2.1 Dendrites, Synapses and Axons

The *dendrites* are the input branches of a neuron, which is either able to excite or inhibit the neuron based on the neurotransmitter it receives on its *synapses*. The synapses are the connections between the output branches,

or *axons*, and the dendrites, that allows electrical or chemical signal to be transmitted. An axon is the main output branch of the neuron, and is able to transport the signal, or action potential, to its synapses. In the dendrites the input signals are integrated, which in turn changes the state of the neuron if a complex set of conditions are met. Firstly, the integrated signal at a spatial position on the dendrite branch needs to exceed a threshold. This threshold is dynamic as the integrated signal needs to be stronger if its further away from the soma. The second condition that needs to be met is the timing of the synaptic conductances that occur at the integration zone [32]. The dendrites can be classified into three zones, the *basal*, *proximal* and *distal* zone. The basal zone contains the basal dendrites that extend along the horizontal axis and connect to neighbouring neurons in the cortical column. The apical dendrite can be divided into two parts, a proximal zone, closer to the soma, and the distal zone which is further away. In the distal zone the amplitude of the integrated signal needs to be higher in order to effect the cell state, due to the longer travel distance. The pyramidal neuron is believed to act as a coincidence detector as its been shown to respond best to simultaneous stimulation in distal and proximal areas of the apical dendrite [32]. Another important feature of the pyramidal neurons is its ability to adapt to change in the temporal input patterns. It able to change functionality due to synaptic plasticity, which is ability to change the strength of the synaptic connection between neurons [32]. As explained in the previous section, the strength of the synaptic connections alters with long-term potentiation or long-term depression.

A.3 Summary

To summaries, HTM theory is derived from properties of the neocortex and the ideas of Vernon B. Mountcastle's single cortical algorithm for information processing. This chapter aimed to familiarise the reader with the structure and functionality of neocortex. First, a gentle introduction to the neocortex was given. Then, we proceeded to the topology and the structure of the individual layers. Next, cortical columns and the neural circuitry was presented, which HTM theory tries to reverse engineer to some extent. Finally, we presented the pyramidal neuron and the functionality of the dendrites. As this is the most abundant neuron in the neocortex, and the inspiration of the HTM neuron's functionality.

ISSN 1653-5146