Hierarchical Temporal Memory – Investigations, Ideas, and Experiments

Thesis · June 2014

DOI: 10.13140/2.1.3380.3848

Stefan Lattner
Sony Computer Science Laboratories (CSL) Paris
67 PUBLICATIONS **417** CITATIONS

JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Hierarchical Temporal Memory - Investigations, Ideas, and Experiments

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Pervasive Computing

Eingereicht von:
Stefan Lattner, BSc.

Angefertigt am:
Institut für Computational Perception

Beurteilung:
Gerhard Widmer, Univ.-Prof., Dipl.-Ing., Dr.

Linz, Juni 2014

Abstract

The goal of this thesis was to investigate the new variant of the Hierarchical Temporal Memory (HTM) of Numenta Inc., which includes the so-called ”Cortical Learning Algorithm”, a connectionist approach to the HTM paradigm. The specific goal was to examine to what extend this HTM variant is able to learn monophonic melodies and chord sequences, and how it performs as a generative model. Another aim was to better understand the HTM, and to describe it formally. Based on the knowledge gained by those examinations, considerations were made on how the HTM could be modified to better handle binary input, to directly process real-valued input, and to build representations of observations in a probabilistic manner.

The HTM was implemented in Java, and it was trained on 20 pop songs. The generated output of the model was evaluated quantitatively and qualitatively. The results show, that the model is able to exhibit a higher-order memory, and that it is able to learn correlations of chords and melody notes. Generated sequences lacked of high-level structures, but showed some mid- and low-level structures. Assessment by listening indicated that the generated output was meaningful, and some of the songs the model was trained on could be recognized.

The HTM, even though the underlying theory seems promising, is still not well specified and therefore difficult to understand and to implement. In it’s current variant, some design decisions were made, which seem to exacerbate the generalization ability of the model and which are difficult to formalize. Some state-of-the-art models exist, which might be better suited than those currently used for the encouraging ideas behind HTMs.

Kurzfassung

Die neueste Variante des Hierarchical Temporal Memory (HTM) von Numenta Inc. beinhaltet den sogenannten ”Cortical Learning Algorithmus”, welcher einem konnektionistischen Ansatz folgt. In der vorliegenden Arbeit soll dieser neue Ansatz untersucht werden. Dabei wird getestet, in welchem Ausmaß das Modell fähig ist, einstimmige Melodien mit Akkordfolgen zu lernen und zu generieren. Ein weiteres Ziel der Arbeit ist es, das Modell besser verstehen zu lernen, sowie es formal zu beschreiben. Basierend auf den daraus gewonnenen Erkenntnissen werden Verbesserungsvorschläge gemacht, welche das Modell befähigen sollen, sowohl binären Input besser zu verarbeiten, wie auch reellwertigen Input zu akzeptieren. Darüber hinaus werden Überlegungen zum probabilistischen Modellieren von Trainingsdaten angestellt.

Das HTM wurde in Java implementiert, an 20 Popsongs trainiert, und die generierten Ergebnisse wurden sowohl quantitativ wie auch qualitativ evaluiert. Die Ergebnisse zeigen, dass das Modell erfolgreich Sequenzen von höherer Ordnung zu lernen im Stande war und dass Abhängigkeiten von Akkorden und Melodienoten modelliert wurden. Die generierten Musikstücke wiesen zwar keine high-level Strukturen auf, aber sowohl mid- als auch low-level Strukturen waren zu erkennen. Beim Hören der Stücke fiel auf, dass das implementierte Modell grundsätzlich sinnvollen Output liefert, und manche der Stücke mit welchen das HTM trainiert wurde, wiedererkannt werden konnten.

Auch wenn die dem HTM zugrundeliegenden Ideen sehr vielversprechend sind, ist es in der vorliegenden Form nicht ausreichend beschrieben und deshalb schwer zu verstehen und zu implementieren. Manche Design-Entscheidungen scheinen dem Generalisieren des Modells entgegenzuwirken, und manche Mechanismen sind schwer zu formalisieren. Das grundsätzlich sehr gute Prinzip hinter HTMs könnte vielleicht mit schon bestehenden Modellen eleganter umgesetzt werden.

Acknowledgements

This work was supported by the project Lrn2Cre8, which is funded by the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET grant number 610859.

Contents

1 Introduction

In 2011, I found a white paper of Numenta called ”Hierarchical Temporal Memory including HTM Cortical Learning Algorithms” [Hawkins, 2011]. This happened during my considerations about how a Machine Learning (ML) model should be designed in order to be able to autonomously learn and produce music. I had already discovered some basic properties of my virtual model. A prediction of any musical event has to be theoretically based on knowledge about the whole song. A note in the melody depends on what chord is currently playing and what chords were played, chords depend on scales, scales depend on the genre, and so on. In addition, music is usually well structured. Repetitions of small and larger parts lead to low-level, mid-level and high-level structures, which should also be covered by the model. Due to the obvious fact that human brains are well-suited for the perception of music, I was also interested in how we perceive music and what correlations between musical structures and human perception are to be found.

With many such considerations in my mind, I read the before-mentioned white paper. I was enthusiastic about what the HTM model could provide me with, because it was very promising in terms of my requirements. Therefore, I started an implementation in Java, which was tough, because even though the report was quite extensive, some processes of HTMs are difficult to understand and some details about how to implement an HTM were missing. I added something new, dropped something proposed, doubted HTMs, constantly trying to get the big picture. This thesis is a report about what I found out about the model and about its ability to create music.

In Chapter 2 an introduction to the HTM is given. I describe how the HTM compares to other ML models and how it was developed in an evolutionary process throughout the last decade. In addition, the basic mechanism of the HTM including the Spatial Pooler (SP) and the Temporal Pooler (TP), the two main modules of an HTM, are presented.

In Chapter 3 of this thesis, the SP is introduced. The SP is a data compression mechanism, which is used in HTMs to extract features in an unsupervised way and to build abstracted representations of input data. Those are then used in the TP to construct temporal sequences of such abstracted representations.

Chapter 4 describes the TP, introduced in [Hawkins, 2011]. Some modifications to the proposed method were necessary to sufficiently perform the task of learning and generating music (see Section 6.2).

For an evaluation of the model, I trained an HTM on musical sequences, like it has been done in [Maxwell *et al.*, 2009]. The paper of Maxwell et al. was based on an older HTM architecture than that introduced in 2011. The goal of my experiments was to evaluate the new architecture in terms of its ability to generate meaningful output after exposure to different musical sequences of monophonic melodies and chords (see Chapter 6).

In Chapter 7, I introduce ideas for improvements of the Numenta SP. Those improvements are based on the geometrical approach to the SP introduced in Chapter 3. My motivation was to enable the SP to better handle binary input data (see Section 7.1), to directly process real-valued input (see Section 7.2), and to build representations of observations in a probabilistic manner (see Section 7.3). The suggestions in that chapter are theoretical considerations for improvements which should be regarded as starting points for further examination and were neither implemented nor tested.

2 Hierarchical Temporal Memory (HTM)

2.1 Introduction

Many real-world problems are laid out in two basic directions: space and time. However, space is usually multidimensional, and a space dimension differs significantly from the time dimension. Data in space is usually distributed in a finite range whereas data in time is theoretically infinitely extended. Additionally, in most cases the time dimension is more uniformly populated than a space dimension and close adjacent data points are mutually highly dependent. Statistically, those properties of time result in models where the prior probabilities can be neglected if a neighboring state is known. In conformity with the HTM terminology, we will denote static Machine Learning (ML) models which attempt to detect correlations in the input space as *spatial* models, whereas temporal sequences are represented within *temporal* models.

Most ML concepts model either spatial or temporal dependencies. There are numerous spatial classifiers like Decision Trees, Artificial Neural Networks, Support Vector Machines, Bayesian classifiers, Rule-based classifiers or clustering methods. Even if the input to such classifiers can consist of multiple time steps, when it comes to proper statistical modeling of time, the set of appropriate models is smaller. Dynamic Bayesian Networks like Markov Models and connectionist approaches like Recurrent Neural Networks are the most popular techniques in that scope.

Dictionary based methods (DM) for signal processing [Sturm *et al.*, 2009] should also be mentioned here, because even though they do not model time through statistical dependencies of neighboring codebook entries or atoms, time can still be laid out in a distinct dimension and is considered differently at processing. It is also interesting to note that DMs share some properties with the Sparse Distributed Representation (SDR) approach of HTMs. In particular, an HTM unit can be seen as an equivalent to a codebook atom, and a set of HTM units can be used to reconstruct data like in DMs (see Section 3.3.2).

Restricted Boltzmann Machines (RBMs) with deep learning are also very similar to HTMs [Hinton *et al.*, 2006]. As in HTMs, RBMs learn features in an unsupervised way, information is stored distributed over several units and it is possible to pile up multiple RBM instances to form a hierarchy. A major difference between those concepts is that the RBM is an energy based model with probabilistic foundation, whereas HTM units cover the feature space in a geometrical manner. But the main difference is that HTMs model time explicitly, while RBMs do not (see Section 2.3.2).

Another approach related to HTMs are Hidden Markov Models (HMMs), which have some similarities to HTMs with respect to their temporal model. From that point of view, an HTM can be seen as hierarchically stacked HMMs, where a sequence recognized by a child HMM represents a single observation for its parent HMM.

The actual strength of HTMs comes from their hierarchical topology, which enables them to create spatio-temporal representations of multiple input sequences at different levels of abstraction. The probability of an event to occur does not only depend on its previous state and an observation, but on the state of the whole model, which is determined by all input sequences and a longer past. HTMs can be trained in a supervised and unsupervised manner. It is possible to reconstruct data from a trained model, to classify streams of input data, and to predict based on the current state of the model.

In practice, there are still a few problems to solve until HTMs are slim, compact and easy-to-use models. Even though the main biologically motivated mechanisms of HTMs are understood and published, many details on how to actually implement an HTM are missing in most publications and research is still ongoing. This thesis is a contribution to that process. I incorporated many of my ideas on how to improve, extend, and implement HTMs to inspire researchers for further investigations. Another important objective of this thesis is to better understand how and why HTMs work. In contrast to Numenta, I have no requirement to stick with biological plausibility. This makes it possible to generalize, for example, the mechanism of the Spatial Pooler (SP) (see 7.1.3). However, Numenta is constantly improving its model and does not provide many details about this process. Therefore it might be that some of my descriptions are based on wrong assumptions due to missing information, or that some problems I solved are already solved by Numenta in a different way. Finally, note that not all of my proposed methods were implemented and tested, even though all of them were thought through very carefully.

2.2 Evolution of the HTM

When Jeffrey Hawkins published his book ”On Intelligence”[Hawkins and Blakeslee, 2004], he already knew which basic mechanisms should be involved in an HTM. Those were derived from his research on the neocortex. Since then, HTMs have been subject to constant evolution. The basic idea of Jeff Hawkins was developed further by Numenta Inc. and the corpus of papers available to HTMs was slowly growing. In 2005, two papers were published [George and Hawkins, 2005a; George and Hawkins, 2005b], in which the first attempts to model the HTM based on Bayesian inference were documented. One year later, Numenta updated the community with a quite comprehensive, yet not scientific technical report where the basic concepts were explained in a more detailed manner and compared to 2005's publications, Jeff Hawkins apparently had a much clearer picture of what an HTM could be [Hawkins and George, 2006]. That paper motivated the first researchers outside of Numenta to experiment with HTMs [Thornton *et al.*, 2008; Schey, 2008; van Doremalen and Boves, 2008].

In 2008, the first implementation of an HTM, "NuPIC" was made available which unfortunately was not well documented and poorly supported by Numenta. In 2013, just before this thesis was finished, the Cortical Learning Algorithm (CLA), which is the main method this thesis is focussing on, was added to the NuPIC project. Additionally, Numenta launched a program to better support the open source community. Now, there are mailing lists available, and the documentation of the code improves. This thesis is based on information available before CLA was implemented in NuPIC. Thus, my implementation of the TP differs from that in NuPIC in that in its general form, my TP has a second-order memory, whereas in NuPIC variable-order chains are created. I found a workaround, which yields variable-order memory, but lacks of the possibility to train the HTM online, and it connects only single units forward in time. Especially the ability to learn variable-order chains of multiple units per time step is a strength of the NuPIC implementation and HTMs in general, and the disability to do so is a major drawback of my variant.

One reason for the hesitant response of external contributors might have been a missing comprehensive formalism for the model. However, in 2008, Dileep George [George, 2008] published a quite comprehensive PhD thesis about HTMs, and in 2009 an article towards a mathematical theory of cortical micro-circuits [George and Hawkins, 2009] was published, which could have led to a breakthrough of the model. Apparently this was not the case even though it included a thorough formalization of their state-of-the-art model. A possible reason may be that the way the SP and the TP was designed was still not very elegant. In the last fundamental white paper which was published in 2011 [Hawkins, 2011], CLA was introduced, a neuronal approach which is elegant and compact enough to deserve acknowledgement. But the paper again lacked formal description. It seems like Numenta has not yet found its target community. Computer scientists focus rather on proven models like the RBM and for others, the model is too complicated. I think, as long as Numenta cannot catch the attention of computer scientists, research on that model will not make the progress it deserves.

In any case, the HTM concept remains very promising for ML tasks which deal with sequential real world problems with potentially huge feature spaces. Especially the before-mentioned publication of 2011 gives insight into a pretty mature variant of an HTM, where the SP and the TP are strongly connected so as to form a

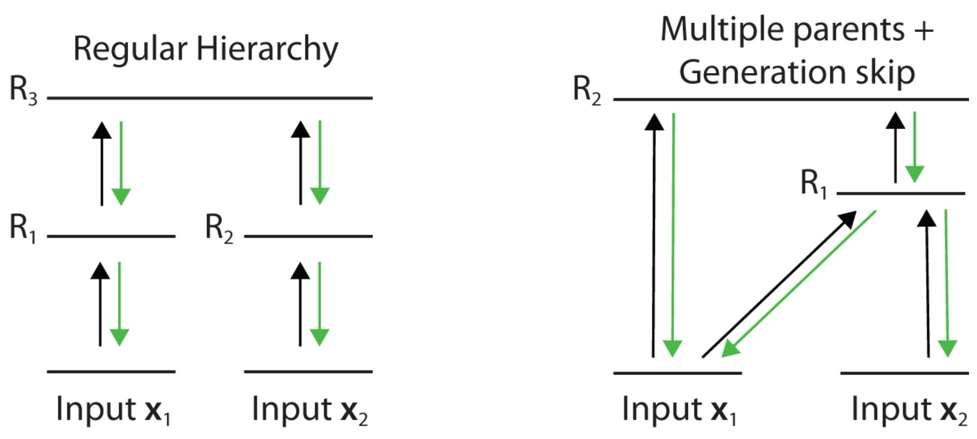


Figure 2.1: Possible topologies of an HTM.

compact model. But assumptions have been made to simplify the HTM so that it moved away from the 2005 variant which was backed up by solid bayesian theory.

Attempts were made to summarize the whole concept of HTMs [Fernández *et al.*, 2011], but as far as we know no such paper is available which incorporates those findings published in Numenta's technical report of 2011. Thus, an interested researcher has to browse through a patchwork of papers, each revealing partial information fragments, and cannot be sure that he understands the model as Numenta does. As stated above, the novel model is not yet supported by a comprehensive formal description, which is why in this thesis the attempt should be undertaken to amend this situation.

2.3 The Model

In this section, I will summarize the basics of HTMs, with strong focus on the latest publication of Numenta [Hawkins, 2011], where a neuronal approach was proposed.

An HTM consists of several hierarchically stacked and mutually connected regions (see Figure 2.1). A parent region can have one child or multiple children, and a child can be either another region or a direct input to the HTM. As output both direct inputs and child regions provide binary vectors $\mathbf{x}_k \in \{0,1\}^N$ to their parent region, whose actual *feed-forward* input $\mathbf{x} = \{\mathbf{x}_1, \ldots, \mathbf{x}_L\}$ is then a concatenation of the output vectors of its children.

Each region consists of a fixed number M of units $\mathbf{c}_j$, where M is to be chosen proportional to the expected number of clusters in its input space (see Figure 2.2). Each unit of a region acts as a feature detector and is connected to its individual randomly initialized subset of input bits. The size of that subset is usually a fixed fraction of the size of the entire input vector. In addition, during training those connections can be cut or re-established. Units can be *active* or *inactive*, depending on the currently observed input $\mathbf{x}^t$. The set of *active* units $S^t = \{\mathbf{c}_a\}$ constitutes the *state* of a region at time step t. A state represents $\mathbf{x}^t$ in an abstracted and compressed way, and it is possible to approximately reconstruct $\mathbf{x}^t$ given S^t. Computing S^t from $\mathbf{x}^t$ is called Spatial Pooling (SP). The region's binary output vector corresponds to a state in that each on-bit represents an *active* unit and each off-bit represents an *inactive* unit. For a more detailed description on the SP, please refer to Section 2.3.1.

In addition to having vertical input connections, during training *active* units get connected forward in time horizontally within a region.*Active* units of time step t get connected to *active* units at time $t+1$. In fact, it is slightly more complicated, because each unit has subunits which actually connect to each other, but for now this is a valid picture. By doing so, as a set of *active* units constitutes a state S, temporal sequences of states $C = \{S^{t-d}, \dots, S^t\}$ are constructed. For a more detailed description on this Temporal Pooling (TP), please see Section 2.3.2.

Usually, HTM training data is sequential, like a melody consisting of musical notes, which are temporally ordered and input one after another. It is also possible to provide an HTM with more than a single input sequence, for example with a sequence of musical notes and a corresponding chord progression. The goal of an HTM is to learn statistical dependencies within and between such sequences in order to predict possible next states of any of those.

As described above, each region converts its current feed-forward input $\mathbf{x}^t$ into an internal state S^t. Furthermore, it stores short sequences C_l consisting of temporally ordered internal states. During an online training process, stored sequences are constantly compared and updated based on the currently observed state sequence and are then used to predict the next state, which is constituted by a set of *predicted* units. Thus, units do not only depend on the observed input in being *active* or *inactive*, but they can also be *predicted* following a region's internal state.

The *feed-forward* output of a region is the state sequence C_o^t which is a representation of the currently observed input sequence. It is represented as a binary output vector, where each on-bit corresponds to an *active* or *predicted* unit, an off-bit means the corresponding unit is neither *active* nor *predicted.* That output constitutes the input $\mathbf{x}^t$ for the region above which again transforms that sequence into a state S^t. Then it again constructs sequences of those states, which can be thought of as learning sequences of sequences. That way, the higher up a region in a hierarchy, the longer the timespan its sequences represent. This has two implications. Firstly, regions at higher hierarchical levels have a longer memory. Secondly, the internal states of regions higher up are more stable with respect to time. That is because a child region outputs a short sequence consisting of the current state and predicted states, while it steps through the states of a sequence. Therefore, its output changes more slowly than its input which results in more slowly changing internal states of the parent region. This mechanism leads to *temporal invariance* (see Section 2.3.2.1).

What we have looked at so far was *feed-forward* propagation in an HTM. In order to predict the next state of a sequence, it is necessary to execute *feedback*, too. As stated above, a region builds only short sequences of its internal states. Based on a short memory, it might be uncertain about what to predict. Its parent region, however, has a longer memory which can be used to reduce the set of possible next states. As stated above, it is possible to approximately reconstruct an input $\mathbf{x}$ given a state S. In the same way, also predicted states can be used to reconstruct input data. The reconstructed prediction of a parent region results in *feedback-predicted* units of a child region.

This gives rise to the question why we even need a child region, if its parent knows better anyway. The answer is simply that internal states get computed with a lossy data compression. Therefore, although the parent region stores longer timespans, its view onto the actual data is blurred. A child region and its parent region together combine their knowledge and make a better prediction than if the child region were to predict by itself. Combining knowledge of two regions can be done by considering only such units of the lower region as *predicted*, which are both *predicted* and *feedback-predicted* (see Section 6.2.3).

Note that a region might have more than one child, resulting in an internal state which constitutes a model of statistical dependencies between all of them. A sequence which is constructed based on such states can model, for example, how different musical events change together over time. Regarding *feedback*, a parent region which combines the output of multiple child regions provides each of them with context information of all branches it combines. A region which should predict the next musical note can so take advantage of, for example, the information about which chord is likely to occur next.

2.3.1 The Spatial Pooler (SP)

SP is an unsupervised clustering method. A region converts the input pattern $\mathbf{x}^t$ it currently observes into an internal state S^t. Such a state represents a cluster, a pool to which the current input pattern belongs. S^t can also be considered as an abstracted representation of $\mathbf{x}^t$. As S^t is computed using the SP, we will refer to it as *spatial state*. The training of an SP will be called *spatial training* and correlations between multiple regions spatially pooled by a parent region without taking into account any temporal relationships will be referred to as *spatial context*.

Consider Figure 2.2, where the bitmap of the letter E is provided as an input pattern. There are several different manifestations of bitmaps which could show an E, but all of them can be put into the same pool with the label ”E”. The actual

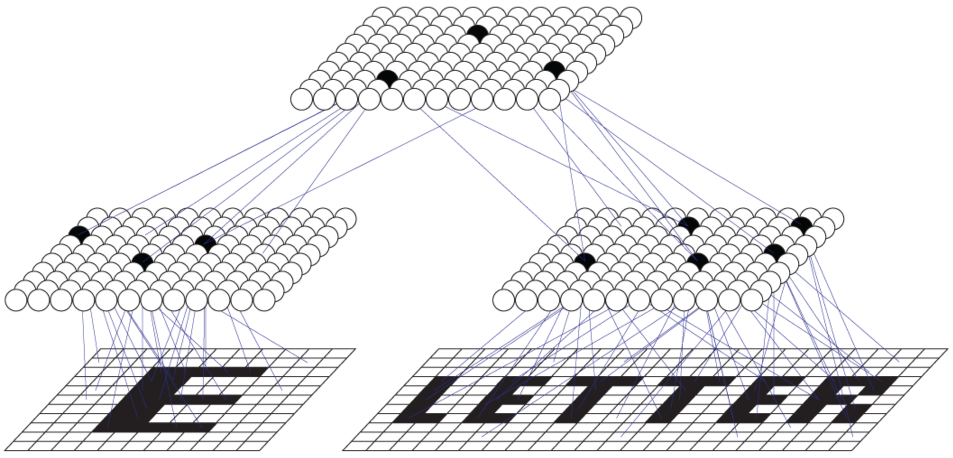


Figure 2.2: Schematic illustration of the basic functioning of the SP.

goal of SP is to form such pools in an unsupervised manner and to assign each input pattern to the pool it belongs to. A pool has a specific manifestation, which is an abstracted representation of the input. Regions in higher hierarchical levels can use those abstracted representations, which leads to a generalization capability of the whole HTM. A representation can also be used to reconstruct an input. Reconstruction reveals an average over all input vectors assigned to a specific pool. We will refer to such an average vector as *archetype*.

In the neuronal approach to HTMs, Sparse Distributed Representations (SDRs) are used for pooling. Consider our example in Figure 2.2. A region consists of multiple units $\mathbf{c}_j$, and each of them is connected to random input bits $x_i \in \mathbf{x}$, some of them are on-bits (black) and others are off-bits (white). If an input pattern $\mathbf{x}^t$ is provided to a region, those n units which maximize an overlap function are the *winning units* $\mathbf{c}_a, a = \{1, \ldots, n\}$ and become *active*. Maximizing an overlap function can be considered as having the most connections to on-bits, compared to all other units. In other words, the overlap value determines for each unit, how much it is specialized on a certain on-bit combination.

For each connection there is a *permanence value* determining if the connection is actually established. If that value is below a threshold, the unit disconnects from the corresponding input bit. During training, each winning unit gets further specialized onto the on-bit combination responsible for its activation by decreasing the permanence value at connections to off-bits and by increasing that value for connections to on-bits. That way, over several training iterations, a winning unit cuts connections to off bits and might re-establish connections to on-bits. Because units are sparsely connected to the input, each unit models the statistical dependence of only a part of the whole input pattern. As a result every time a similar combination of on-bits is observed, certain units become *active*. The whole set of *active* units forms S^t, an SDR of the pool the pattern $\mathbf{x}_t$ belongs to.

This method executes pattern recognition and data compression simultaneously, and a region's state can conveniently be transformed into an output pattern, which is yet another input pattern for its parent regions (see Figure 2.2). That output pattern is simply a binary vector, where all bits are off, except those corresponding to *active* units $\mathbf{c}_a$. By doing so, data get ever more compressed and abstracted. At any desired level of abstraction, the output of multiple regions can be combined to learn statistical dependencies between pools of different direct inputs. For example, in Figure 2.2 the inputs "E" and "LETTER" are transformed into SDRs at the first hierarchical level, where dependencies of input bits are represented. Those representations are then combined into yet another representation at the highest hierarchical level, which models the dependency between pool "E" and pool "LETTER".

This example shows two aspects of HTMs. Firstly, inputs to HTMs can be rather low-level, because they get compressed very efficiently and actual dependencies between inputs can then be computed based on those compressed representations. Secondly, an HTM could be trained in a supervised way, too. Even though "LETTER" is not a strict label to the image "E" but rather an additional feature, it helps to separate the E pool from pools of other things looking similar to an E, for example a rake. However, instead of the input vector showing "LETTER", there could be a vector of the size of the alphabet where each bit of such a vector corresponds to a certain letter. For each letter exposed to the HTM during training, the corresponding bit could then be turned on and such a "label bit" would allow for actual supervised training.

As stated above, the actual purpose of learning SDRs from input data is to use them for building sequences in order to predict. However, any internal spatial state of a region can be easily reconstructed (see Section 3.3.2.2), and I recommend to compare input patterns to reconstructed patterns in order to monitor the learning progress. Moreover, it is probably possible to sample from a trained HTM based on Naive Probabilistic Sparse Distributed Representations (NPSDRs) (for a possible sampling process, see 7.5). Such methods were not yet proposed by Numenta. They are a result of my own research in HTMs.

2.3.2 The Temporal Pooler (TP)

Temporal pooling is done within a region and has the purpose of building sequences of the region's states. Those sequences are then compared to current input sequences and used to predict future input.

Figure 2.3 shows an HTM region, which is already spatially trained on the letters "E", "V", and "N". For simplicity we assume that a spatial representation of any

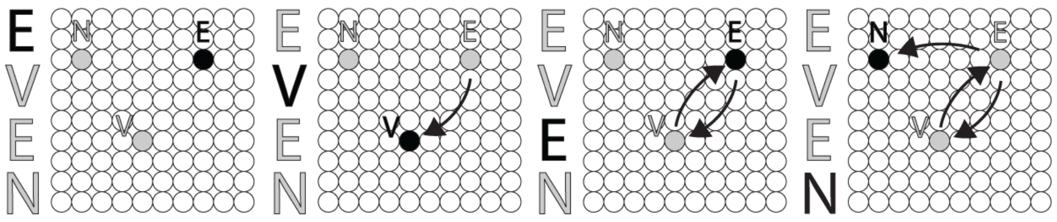


Figure 2.3: States of the sequence "E-V-E-N" as a Markov chain of order 1.

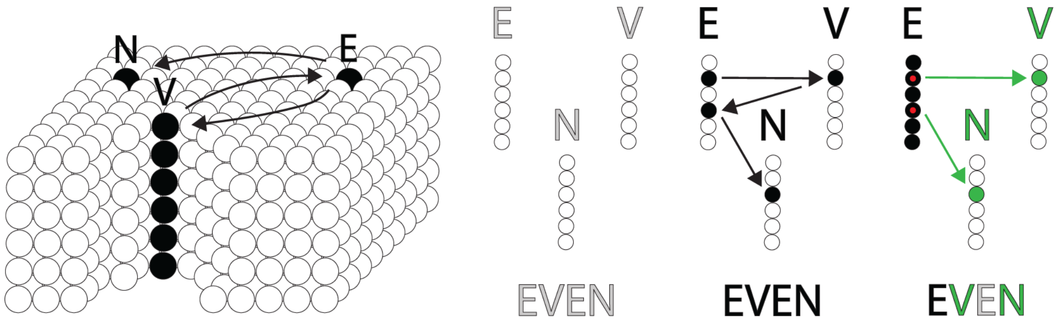


Figure 2.4: On the left side: States of the sequence "E-V-E-N" in a region with units extended to columns. Right: Units responsible for letters E, V, and N temporally connected, which results in a representation of the sequence ”E-V-E-N” as a higher-order Markov chain.

letter consists of only one *active* unit, $n = 1$. If the sequence ”E-V-E-N” is observed by the region, one unit after the other becomes *active*. As shown in the illustration, it is possible to simply forward-connect *active* units corresponding to their order in time. The currently *active* unit and the previously *active* unit could then be fed forward to the parent region, as if both units were simultaneously *active*. Even though this output can be considered as a representation of a very short sequence, it does not carry information about the order of those two states.

To overcome those limitations, Numenta proposed an extension to that naive approach. Let’s consider Figure 2.4, where the whole region gets extended by an additional dimension. A unit is now a column of sub-units, but the functionality of a unit with respect to SP remains the same. As we can see, instead of assigning a unit to each letter in Figure 2.3, a column is now assigned to each letter. The sequence ”E-V-E-N” can then be stored by using the additional dimension to represent the order of occurring states, as well as to increase the possible length which can be represented. Note that as the letter ”E” occurs after ”V”, a unique node gets activated which represents ”E after V” and the node active in the ”N”-column represents ”N after E after V”.

On the very right side of Figure 2.4 a whole column is active and predicts two possible next states. This is an elegant way of comparing multiple stored sequences to a currently occurring sequence at once. In this example we assume that none of the sub-units of the ”E”-unit was *predicted* in the previous time step $t - 1$, which is why at time step t all sub-units of the column became active as the unit itself became active. If a sub-unit of the column were *predicted* at $t-1$, only that sub-unit would become *active* at time step t. All *active* sub-units forward predict in time and if a whole column is active, based on the current state all possible next states get *predicted* without considering the past. Therefore, it is even possible to detect a sequence without having it started from the beginning. In this example, if after the ”E” an ”N” occurs, the subunit ”N after E after V” becomes *active*, because it was *predicted*. This can be seen as having detected the sequence ”E-V-E-N” without actually having observed the first two letters.

As the output for a region with that TP, Numenta proposed to forward the currently *active* sub-unit plus the *predicted* sub-units, as if they were *active*, too. However, my own tests have shown that combining the current state plus predictions is not unproblematic for certain problems, when it comes to incorporating feedback information into a region’s prediction (see Section 4.1.4.1). Instead, I propose to either output the current state and the previous state or to exclusively output the current state (see Section 2.3.2.1).

Feedback is then processed by intersecting the predictions of all regions of a branch of the HTM hierarchy. If a parent region predicts state S_1 and state S_2 and its child region predicts state S_2 and state S_3, the resulting prediction is state S_2. For further explanations concerning the TP, see Chapter 4.

2.3.2.1 Temporal Invariance

Temporal invariance is a very important property of HTMs. Jeff Hawkins points out that this ability is also vital for processing data in the neocortex [Hawkins and Blakeslee, 2004]. If a human being is exposed to a known temporal sequence, in higher layers of the neocortex neuronal activation patterns can be observed which tend to stabilize throughout the whole sequence. Those patterns provide temporal context information to lower regions. In general, temporal invariance enables the brain to organize temporal experiences in a hierarchical way.

In HTMs, short sequences get transformed into spatial states by combining the current state plus the prediction or previous states in the feed-forward process. In regions higher up the hierarchy, states are getting more and more stable in time, which leads to a memory of a potentially very long time span. That way, in the feedback process, child regions are not only provided with context information about branches of the hierarchy they are not part of, but also with context information about what happened in the past. Note that in the feed-forward process sequences are fed up by considering all states of the sequence as active without explicit order information. That way, the input to a parent region is only slowly changing which helps to introduce temporal invariance. However, this is also a big disadvantage considering that it is impossible to reconstruct exactly the next state of a sequence, because the parent region's prediction is a mixture of several states of the sequence.

Direct inputs which do not change over a certain timespan, while other direct inputs do change, help to stabilize states in time, too. This would be the case, for example in Figure 2.2, if the feature ”LETTER” remained stable over time, while the other direct input iterated over several different letters. In this case, some units of the upper-most region could remain *active* over several iterations. If, for example, a lot of symbolic musical events were input into an HTM with a high temporal resolution, over most iterations no input would change. A musical scale event remains stable over a very long timespan, and so a genre event would do. Chords are changing more often, but do not change so frequently as, for example, musical notes. Such input data, where many event types overlap temporally in their occurrence yield states of regions in an HTM that are quite stable with respect to time. In this case, the output of a region might consist only of the current state, without the prediction while still maintaining the temporal invariance property.

3 Spatial Pooling (SP)

3.1 Introduction

What Numenta refers to as ”Spatial Pooling” is a method derived from Vector Quantization (VQ) [Gersho and Gray, 1991], a data compression technique similar to clustering methods, like the popular k-means algorithm or Self Organizing Maps (SOMs). The goal of VQ is to approximate the probability density of potentially high dimensional data via so called Quantization Centers (QCs). Similar to Cluster Centers (CCs) in clustering methods, QCs partition the space into subgroups, where the number of QCs in a certain area is proportional to the density of data points in that area. In contrast to clustering algorithms, the number of QCs in VQ is much higher than the number of CCs in clustering and it is not necessary to estimate the expected number of clusters in advance. VQ training algorithms, however, are very similar to those in clustering but due to the high amount of QCs, the risk of getting stuck in a local optimum is smaller. For sake of simplicity, we will sometimes refer to VQ as clustering, because in some explanations it might catch one’s imagination better than using terms like density estimation or quantization.

Each HTM region consists of multiple units, which can be seen as a set of QCs, also referred to as codes. They are trained on the respective region’s input data where they distribute themselves geometrically over the input space in order to model important correlations between dimensions. If the number of units is chosen smaller than the size of the input vector, the state of a region given an observation is a compressed representation of the observation. Furthermore, such representations can be connected to temporal sequences by the TP. In addition, they are passed on to higher regions which, in turn, further compress that information.

Note that the input to a region is usually a concatenation of more than one sensor inputs or a concatenation of outputs of two or more child regions. Compressed representations of such combinations unveil correlations of different inputs. That is useful if sampling should be performed based on a stack of trained regions. Given the state of a region which combines the outputs of multiple child regions, information about the plausibility of each of the children’s states, given the other children’s states can be gained.

In this chapter, a short review will be given on how VQ was utilized for HTMs. Note that the SP is designed for binary input vectors. Interestingly, Numenta neither published a method to process real valued data nor described how the SP could be used to probabilistically model dependencies between spatial input vectors and their assigned QCs as it is the case in similar models like the RBM. However, this would be necessary to thoroughly formalize HTMs in a probabilistic way on both the spatial and the temporal level.

Recently, Numenta published a white paper introducing a neuronal approach to HTMs, where Sparse Distributed Representation (SDR) is used instead of common VQ [Hawkins, 2011]. Therefore, in this chapter a bridge will be built between early attempts of "Spatial Pooling" and the current state of Numenta publications including SDR and the "Cortical Learning Algorithm" (CLA). This will be done based on established formalisms to re-generalize the simplified binary model proposed by Numenta. This formalisation will naturally show how HTMs could be applied to real valued data, which will be described in Chapter 7.

3.2 Vector Quantization

Vector Quantization (VQ) is a simple method which can be seen as a starting point for our considerations on how to formalize and extend the SP currently proposed by Numenta. In VQ as well as in clustering, two aspects have to be taken into consideration. One aspect is the VQ method, which describes the basic mechanism to initialize and train the VQ model. The other aspect is the distance function $D(\cdot)$, the central yet exchangeable function in every VQ method.

Let $R = \{C, S\}$ be a region with a codebook C and a set of possible states S. The $M \times N$ matrix $C = \{\mathbf{c}_1, \dots, \mathbf{c}_M\}$, referred to as code book, is the concatenation of QCs, stored in that region. A QC is also called code or code book entry $\mathbf{c}_j = \{p_{1j}, \dots, p_{Nj}\}^T$, where p_{kj} is the value of the k-th dimension of a QC $\mathbf{c}_j$. Furthermore, let $\mathbf{x} = \{x_1, \dots, x_N\}^T$ be the input vector to a region, also referred to as training instance or data point, where x_i is the value of the i-th dimension of that vector.

Let $D(\mathbf{x}, \mathbf{c}_j)$ be the pairwise distance function of the input vector $\mathbf{x}$ and QC $\mathbf{c}_j$. If an input vector $\mathbf{x}$ is exposed to a region and the region decides for n units $\mathbf{c}_a$ to be the closest, we say that $\mathbf{c}_a$ are *active* and the region is in state

$$S = \{c^n(\mathbf{x})\} = \{\mathbf{c}_a\} \subset C, \tag{3.1}$$

where $c^n(\mathbf{x})$ are the n QCs $\mathbf{c}_a, a = 1, \dots, n$ that minimize the distance function $D(\mathbf{x}, \mathbf{c}_j)$ and $S \subset C$ is a matrix containing all *active* QCs $\mathbf{c}_a \in C$. Note that most clustering methods have just one QC assigned to each input vector which were written as $c^1(\mathbf{x}) = c(\mathbf{x})$. However, we formalize state S of a region in a general way, because an HTMs state typically consist of more than one QCs, as we will discuss later on.

In the neuronal model of Numenta a code is represented by a unit $\mathbf{c}_j$ and the unit's edges p_{ij}, which point to some random input dimensions x_i. In that model, the unit itself represents the code's index and the permanence values p_{ij} of the unit's edges represent the values of the code.

The general goal of clustering is to minimize the overall error which would be caused if data were reconstructed following the information of the code book. Different distortion measures are used, which are the same for VQ as well as for clustering. A very common error measure is the Mean Quantization Error

$$MQE = \frac{1}{|X|} \sum_{\mathbf{x} \in X} D(\mathbf{x}, c(\mathbf{x})). \tag{3.2}$$

This equation makes it clear why the density of QCs should be proportional to the density of data points. That way, commonly occurring data has low error and rare data has higher error, which leads to a minimization of the overall distortion function.

3.2.1 Vector Quantization Methods

At first, we will have a look at different VQ methods which were used, or which could be used with HTMs. In the first white paper of Numenta [Hawkins and George, 2006], a dictionary based clustering method was suggested, without unveiling the actual method which was used. Therefore we will discuss possible methods in this section.

Generally, any VQ or clustering method like k-means, Self Organizing Maps (SOMs), Learning Vector Quantization (LVQ) or Neural Gas (NG) could be sufficient. However, due to certain properties of HTMs, some approaches are to favor. For example, a region which is in state $S = \{\mathbf{c}_a\}$ again acts as an input vector for its parent region. This can be achieved by converting the $M \times N$ code book C into a binary vector of size M, at which all positions are set to zero, except positions a, which are set to one. Due to these inter-region dependencies, a code book should remain of fixed size and it would be cumbersome to adapt that size during training. Following that constraint, the set of appropriate clustering methods is restricted, too. For example, Growing Neural Gas or Growing Self Organizing Maps are not appropriate.

What we will look at for now are approaches where in training and reconstruction each given data point is represented by only the one QC closest to the data point. That is different with SDRs, where several QCs are considered for training the code book and for reconstruction of a data point.

3.2.2 Initialisation

A possible approach to initialization were to fill the initially empty code book as data is observed. Given an input vector $\mathbf{x}$, the closest code book entry $c(\mathbf{x})$ is found by using the distance function $D(\mathbf{x}, \mathbf{c}_j)$ (see Equation 3.1). If that distance exceeds a threshold while the code book is not yet full, a new code book entry is generated. If none of those conditions are met, the closest code book entry is adjusted as described in Section 3.2.3.

That approach, however, bears the risk that the code book is filled up before its set of entries form a representative coverage of the observed data. In this case, if input data of a yet unobserved, potentially small cluster is exposed to a region, the closest code book entry might miss to adapt properly to that cluster, especially if it already covers another, bigger cluster whose data points hold that QC captive. Because of the above-mentioned restriction, namely that the size of a code book should not be increased during training, such an initialization might be problematic.

However, if the code book size or the distance threshold is chosen high enough to receive clusters that cover the input space properly before the code book runs full, compared to random initialization methods for problems where the input space is sparsely populated, the training speed might increase, because QCs tend to be generated at positions where the probability is high that a local optimum is close.

Another approach to initialize a code book is to fill it with random entries, which is a method used by popular algorithms like k-means, LVQ or NG. That way, QCs are statistically equally distributed throughout the input space, whereby small clusters, whose data points occur very rarely, might still be covered. That approach is more general to the one mentioned above and it can be applied to problems where no assumption about the data distribution can be made at the outset.

3.2.3 Training

Once initialized, the code book needs to be trained to better fit the distribution of the input data. This means minimizing a global reproduction error function, like shown in Equation 3.2. Again, the optimal solution for a code book to cover the underlying data is that all local maxima of its probability density function (PDF) are occupied by QCs to best represent archetypes for higher regions to work with. In other words, the density of QCs in the input space should be proportional to the probability density of data points to occur in their respective neighborhood. This condition ensures that denser areas are covered with higher quantization resolution, which is also desirable, because dense areas unveil their underlying PDF in a higher resolution and the goal of a quantization process is to preserve as much information as possible.

HTMs are trained online, which means training instances of theoretically infinite number are processed one at a time. Therefore, online variants of VQ methods are discussed in this section.

3.2.3.1 K-means training with the Online Lloyd's Algorithm

Lloyd's algorithm [Lloyd, 1982] is the standard training algorithm for k-means clustering. We cover that method because it can be seen as the origin of all other clustering methods described in this thesis and helps to better understand them. Let $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_M\}$ be a code book with Cluster Centers (CCs) $\mathbf{c}_j$ randomly distributed in space. In the offline variant, each data point $\mathbf{x}$ is then assigned to its closest CC $c(\mathbf{x})$. Finally, each CC is placed at the mean of all data points assigned to it. Those steps are executed until convergence.

In an online setting, the mean of data points has to be calculated as data is observed. Here, each CC counts how many data points were already assigned to it so that each data point contributes equally to determining the mean. The algorithm is as follows:

---

1. Initialise M cluster centers $\mathbf{c}_1, \ldots, \mathbf{c}_M$ randomly.
2. Declare counters $n_1, \ldots, n_M$ and initialise them to zero.
3. Get a new data point $\mathbf{x}$
4. Determine the closest QC $c(\mathbf{x}) = \mathbf{c}_j^t$.
5. Update the number of points in that cluster: $n_j^{t+1} = n_j^t + 1$
6. Update the cluster center: $\mathbf{c}_j^{t+1} = \mathbf{c}_j^t + \frac{1}{n_j}(\mathbf{x} - \mathbf{c}_j^t)$
7. Go to step 3.

3.2.3.2 Vector Quantization

The training method for Vector Quantization results in a very simple yet effective online training. The idea is that if a new data point $\mathbf{x}$ is observed, its closest QC $c(\mathbf{x})$ is found. That QC is then moved towards $\mathbf{x}$ by a fixed fraction λ of the distance, where $\lambda \in [0, 1]$ serves the purpose of a learning rate factor. The algorithm remains the same as the Online Lloyd's Algorithm (see Section 3.2.3.1), except Equation 6 changes to

$$\mathbf{c}_j^{t+1} = \mathbf{c}_j^t + \lambda(\mathbf{x} - \mathbf{c}_j^t). \tag{3.3}$$

In the latest white paper of Numenta [Hawkins, 2011], QCs are moved towards data points by a fixed factor instead of by a fraction of the distance between the QC and the respective data point (see Equation 3.7). In an equation, this would mean to normalize the distance vector $\mathbf{d} = \mathbf{x} - \mathbf{c}_j^t$ to receive the direction for every dimension the QC should move towards, and then multiplying that vector with a fixed factor δ which controls the step width or learning rate. However, this is done more easily with the neuronal approach of Numenta, which will be discussed in Section 3.3.1.2.

3.3 The Numenta Spatial Pooling

Recently, Numenta proposed a neuronal approach to HTMs [Hawkins, 2011], including SP by using SDRs (see Section 3.3.2). SDRs are very similar to VQ, but instead of having one QC assigned to each data point, with SDRs a data point is represented by a number of QCs in its neighborhood. Moreover, in contrast to common VQ models, in SDRs a QC doesn't model dependencies over all dimensions of the input space, it rather ignores most of them. Information is thus distributed over several QCs and reconstruction can be done by averaging their positions. In this section, we will approach SDRs step by step using the neuronal model of Numenta. In Section 3.3.1 we will describe what sparsity means and how it may lead to higher robustness in pattern recognition. Then we will extend that model to distributed representations (see Section 3.3.2).

3.3.1 Sparse Representations

3.3.1.1 Example

Let's consider a very simple two dimensional example of a trained region R, which is a neuronal VQ model as shown in Figure 3.1. Its units $\mathbf{c}_j$ represent QCs, distributed over the input space. A binary input $\mathbf{x} = \{x_1, \ldots, x_N\}$ of length $N = 2$ may adopt one of four possible states $\{\mathbf{x}_1, \ldots, \mathbf{x}_4\} = \{\{0,0\}^T, \{0,1\}^T, \{1,0\}^T, \{1,1\}^T\}$, which means it can represent one of four different data points in the input space. We also name $\mathbf{x}$ a pattern and $x_i = 1$ an on-bit, $x_i = 0$ an off-bit. The weights $w_{ij} \in \mathbf{w}_j, w_{ij} \in \{0,1\}, \mathbf{w}_j \subset \mathbb{N}^N$ of the edges which connect each unit with all input dimensions specify if unit $\mathbf{c}_j$ is connected to the x_i dimension. Thus, in this neuronal model, the code book is split into units and edges. Numenta uses the terms neurons and synapses instead of units and edges, but we will stick with our terminology because it is more general and extendable. A unit represents the entity QC, which denotes the index of a code in the code book, the actual entry is encoded

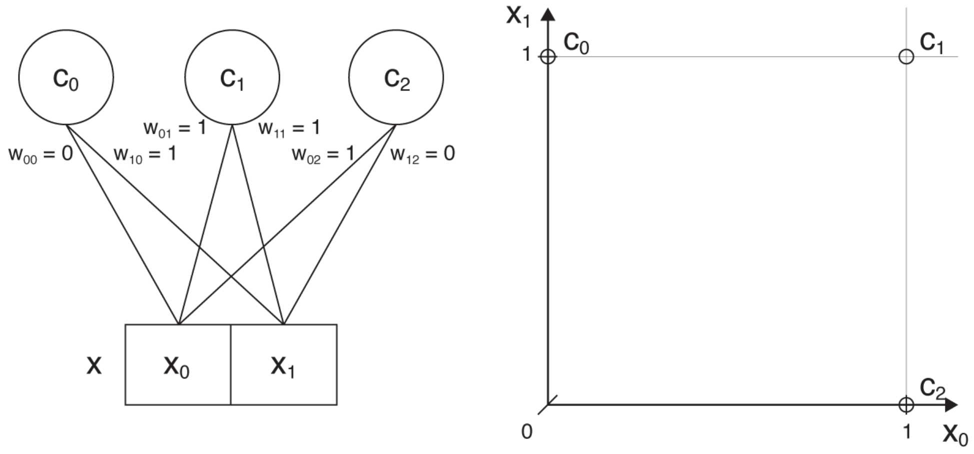


Figure 3.1: Neuronal model of Vector Quantization.

in its outgoing edges. We can still look at units and edges as code book entries (see Section 3.2), where the QC corresponding to a unit $\mathbf{c}_j$ can be defined as a vector $\mathbf{c}_j = \{w_{1j}, \ldots, w_{Nj}\}^T$.

Given an input $\mathbf{x}$, the closest QC $c(\mathbf{x})$ can be obtained with the following equation:

$$c(\mathbf{x}) = \arg\max_{\mathbf{c}_j} O(\mathbf{x}, \mathbf{c}_j), \tag{3.4}$$

where $O(\mathbf{x}, \mathbf{c}_j)$ is a ”closeness” function, which in [Hawkins, 2011] is referred to as overlap

$$O(\mathbf{x}, \mathbf{c}_j) = \frac{\sum_i^N x_i w_{ij}}{\sum_i^N w_{ij}}. \tag{3.5}$$

As we can see, the overlap results from comparing a vector $\mathbf{c}_j$ to the input vector $\mathbf{x}$. For example, consider the model illustrated in Figure 3.1 and let input $\mathbf{x} = \{1, 0\}^T$. The overlap of unit $\mathbf{c}_1 = \{1, 1\}^T$ would yield 0.5, while the overlap of unit $\mathbf{c}_0 = \{1, 0\}^T$ would result in 1.

Following Equation 3.5, an overlap is not a direct counterpart to a distance function in the N-dimensional input space, it is rather a measure which determines how many of connected edges are actually pointing at on-bits. For each QC $\mathbf{c}_j$ only such input dimensions x_i are taken into account, where $w_{ij} = 1$. Thus, each QC is specialized on a input subspace in which the overlap is a proper equivalent to the Hamming Distance, divided through the number of dimensions of the respective subspace, in order to make the result of QCs with a different number of connected edges comparable.

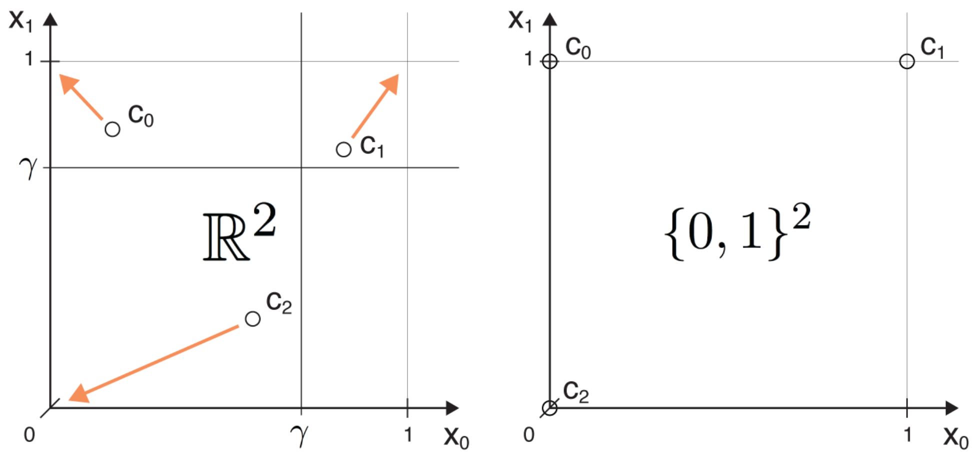


Figure 3.2: Left: The positions of QCs in the real-valued space are defined by their corresponding permanence values. Combined with threshold γ, positions in the binary space are determined. Right: The position of QCs in the binary space are set by their weights, which snap to 0 or 1 according to the corresponding permanence values and threshold γ.

Note that the input $\mathbf{x} = \{0, 0\}^T$ results in an overlap of 0 for every unit. Furthermore, a QC specialized on such an input would cut all connections and would have to ”live” in an empty subspace. In Equation 3.5 the denominator would yield zero. Therefore, such an input is not valid.

3.3.1.2 Training the model

If an edge does not receive enough stimulation during training, it gets disconnected from its input dimension. Units are then sparsely connected whereby the robustness agains noise gets increased. For example, in Figure 3.1 the unit $\mathbf{c}_0$ is sparse in that it ignores input dimension x_0, while the unit $\mathbf{c}_2$ ignores dimension x_1.

In the example above we assumed an already trained model. Now we look at how this training can be accomplished. For that, let $p_{ij} \in \mathbf{p}_j, p_{ij} \in [0, 1], \mathbf{p}_j \subset \mathbb{R}^N$ be a permanence value for every edge E_{ij} connecting unit $\mathbf{c}_j$ to the input dimension x_i. If that permanence value exceeds a threshold $\gamma \in [0, 1], \gamma \in \mathbb{R}$, the weight w_{ij} of edge E_{ij} is set to 1 and we say it is connected, otherwise it is set to 0 and is regarded as being disconnected:

$$w_{ij} = \begin{cases} 1 & \text{if } p_{ij} > \gamma, \\ 0 & \text{if } p_{ij} \leq \gamma. \end{cases} \tag{3.6}$$

That way, in addition to the binary input space, through introduction of permanence values a real-valued space is created. Let $\mathbf{c}_j = \{\langle w_{1j}, p_{0j}\rangle, \ldots, \langle w_{Nj}, p_{Nj}\rangle\}^T$

be the such extended QCs, where a tuple $\langle w_{ij}, p_{ij}\rangle$ can be seen as an edge. Permanence values p_{ij} define the position of $\mathbf{c}_j$ in the i-th dimension of the input space. Therefore, we will also refer to permanence values as positions.

In online training, the positions of QCs are adjusted, so that they move towards data points $\mathbf{x} = \{x_1, \dots, x_N\}$ in their neighbourhood. If a training instance is presented to the model, the closest QC is found by using Equation 3.4 and its permanence values get increased at those edges which are connected to on-bits and decreased at those which are connected to off-bits as

$$p_{ij}^{t+1} = \begin{cases} p_{ij}^t + \delta & \text{if } x_i = 1, \\ p_{ij}^t - \delta & \text{if } x_i = 0, \end{cases} \tag{3.7}$$

where δ is the learning rate. If some on-bits of $\mathbf{x}$ tend to occur together, the unit which was the closest in space in the beginning will get specialized on that pattern because its edges connect to on-bits and disconnect to off-bits of that pattern. Once a connection to a bit is cut, the overlap of a unit following Function 3.5 is not influenced by that bit any more. Therefore, patterns can also be recognized under noisy conditions, or if the region is provided with a mixed input[1].

3.3.1.3 Boosting Units

Similar to using sensitivity values in Neural Gas [Martinetz and Schulten, 1991], in Sparse Representations boosting is applied. QCs which are far away from any considerable cluster might very rarely be the closest match given an input, following Equation 3.5. Boosting the overlap function of the respective QC helps bringing it back into the match again. For that, we extend Equation 3.5 by a boosting factor b_j as

$$O(\mathbf{x}, \mathbf{c}_j) = \frac{b_j \sum_i x_i w_{ij}}{\sum_i w_{ij}}, \tag{3.8}$$

$$b_j = 1 - \min(0, \phi(\text{count}(\mathbf{c}_j = c(\mathbf{x}), t) - \omega \text{count}(\mathbf{c}_k = c(\mathbf{x}), t))), \tag{3.9}$$

$$\text{count}(e, t+1) = \text{count}(e, t) * (1 - \lambda) + \text{eval}(e, t) * \lambda, \tag{3.10}$$

$$\text{eval}(e, t) = \begin{cases} 1 & \text{if } e, \\ 0 & \text{if } \neg e, \end{cases} \tag{3.11}$$

[1]Two input patterns, each belonging to another archetype can be mixed with a bitwise OR operator. An example would be a binary picture showing a mouse and a cat instead of only a mouse or only a cat.

where $\text{count}(e, t)$ is a sliding average function at time step t over a number $1/\lambda$ of training iterations in which expression e is true and $\mathbf{c}_k$ is the QC which maximizes that function. $c(\mathbf{x})$ is the closest QC given an input as described in Equation 3.4 and ϕ, ω are scaling factors, where ω in [Hawkins, 2011] equals to 1/100.

In other words, a QC gets boosted if, while training, it was a hundred times fewer updated than the most frequently updated QC and gets the more boosted, the less it was updated. Similarly, all permanences of the edges of a QC get boosted if its overlap function exceeds a certain threshold very rarely.

3.3.2 Sparse Distributed Representations (SDRs)

In the previous section, QCs were considered to be sparse in that they did not take into consideration all dimensions of the input space. The subspace a QC was responsible for, was adapted during training, but there was still the goal of completely representing input patterns in one QC each.

Latest VQ methods use Sparse Coding approaches which yielded better results by distributing the representation of input patterns over several codes of a code book [Coates and Ng, 2011]. Moreover, Pentti Kanerva [Kanerva, 1988] developed Sparse Distributed Memory (SDM), a method also used in context similarity analysis of written words [Sahlgren, 2006] which encoded binary patterns in a distributed manner, too. SDRs seem to be a hybridisation of SDM and VQ methods. In Chapter 7, I introduce a possible optimization to SDRs for binary input based on ideas of SDMs (see 7.1).

3.3.2.1 The Numenta SDR Model

In SDRs, input patterns are stored distributed over many QCs. Each QC is responsible for only a subspace of the input space, which is chosen randomly at initialization. In a neuronal point of view this implies that no unit is fully connected to the input vector. Based on an input vector $\mathbf{x}$, the state of a region is derived by searching for a fixed number n of closest QCs, $S = \{c^n(\mathbf{x})\}$ (see Equation 3.1). Using several codes for each data point improves robustness in recognition, because similar patterns will result in a selection of a similar set of QCs.

As stated above, the subspace a unit is responsible for is chosen randomly at initialization. In order to make a fully connected unit sparse, most weights of its edges are permanently set to 0. A set of QCs are again able to properly reconstruct an input pattern (see Section 3.3.2.2). For implementation of the Numenta SDR model, all equations concerning overlap, training, weighting, and boosting do not differ from what was described in 3.3.1.

3.3.2.2 Reconstructing a Pattern in the Numenta SDR Model

The value x_i of dimension i of a stored pattern to be reconstructed is calculated as

$$x_i = \begin{cases} 1 & \text{if } \sum_a w_{ia} \geq \epsilon, \\ 0 & \text{if } \sum_a w_{ia} < \epsilon, \end{cases} \tag{3.12}$$

for all $\mathbf{c}_a \in S$. The threshold $\epsilon \in \mathbb{N}$ has to be one or higher, depending on how many *active* units' connections should point at the input bit x_i to consider it as being an on-bit.

4 Temporal Pooling (TP)

4.1 The Numenta Temporal Pooling

4.1.1 Basic Concepts

The purpose of the TP is to learn transitions between states S^t of a region R. Given is a spatially trained region $R = \{C, S\}$ with a set of units $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_M\}$, and a set of *active* units $S = \mathbf{c}_a$, as introduced in 3.2. For the TP, we extend that region to $R = \{C, S, U, B, E\}$. Subunits $U = \{u_1, \ldots, u_N\}$ are directly linked to units, therefore we say that each unit $\mathbf{c}_q$ has a subset of $U_q \subset U$ assigned to it. Temporal connections in the TP are created between subunits. In addition, there is a set of segments $B = \{b_1, \ldots, b_J\}$ which bundle incoming connections to subunits. We denote B_l as the subset of segments assigned to subunit u_l. Finally, there is a set of directed "horizontal" edges $E = \{e_0, \ldots, e_K\}$, $E \subset U \times B$ and we write $u \to b$ for an edge which connects subunit u with segment b.

Thus, we have units C, where each unit $\mathbf{c}_j \in C$ "has" a set of subunits U_j, each subunit $u_k \in U$ "has" a set of segments B_k, and there is a set of edges E, which connect some subunits to other subunits' segments.

4.1.1.1 Subunits

There are several subunits for a unit $\mathbf{c}_j$. Like a unit, subunits can be *active* or *inactive*. For TP the set of possible states for subunits is extended by *predicted* and *learning*. The set of *active* subunits is written U_{act}, the set of *predicted* subunits is referred to as U_{pred} and U_{learn} denotes the set of all subunits in the *learning state*. We will have a closer look on why we need those states in 4.1.3, for now we acknowledge that they exist.

A subunit has the purpose of instantiating the unit it is assigned to. This allows for representing one unit in different temporal contexts. If a unit becomes *active* due to feed-forward input, depending on the past, a different subunit of that unit will become *active*.

For the SP, all subunits $u_q \in U_j$ have the meaning of "being" $\mathbf{c}_j$. Imagine a unit $\mathbf{c}_j$ as a column of subunits, where the SP takes a bird's eye view of it and cannot distinguish between its subunits. If any subunit is *active* or *predicted*, the SP considers the whole unit as being *active* or *predicted*.

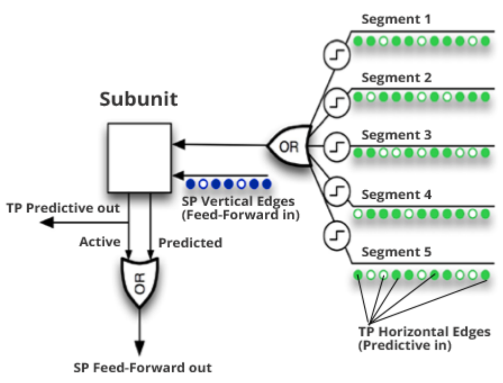


Figure 4.1: Schematic illustration of a subunit. SP: All subunits of a unit share the same feed-forward input while each subunit has a separate feed-forward output which is *active* when the cell is *active* or *predicted*. TP: Each subunit has segments which bundle horizontal (predictive) inputs. If one of the segments is *active* because its input activity exceeds a threshold, the subunit becomes *predicted*. The predictive output is connected to one or more segments of other subunits in the same region and is *active* if the subunit is *active*. Illustration from [Hawkins, 2011], adapted with permission of Numenta Inc.

The TP, however, distinguishes between subunits of a unit. As stated above, it learns transitions between states S^t of a region. But $S^t \subset C$ is a spatial state and the set of all *active* units at time t. For the TP, the corresponding set is $U^t_{\text{act}} \subset U$, which denotes the set of all *active* subunits at time t. As stated above, a subunit can be seen as an instance of a unit. A set of *active* subunits U^t_{act} at a time step t is therefore an instance of a set of *active* units S^t. Thus, we will call U^t_{act} a *state instance*. A state instance is simply a set of subunits. Any state instance of a state has the meaning of that state in a certain temporal context. In order to model transitions between states of a region, we actually model transitions between state instances, because this allows for longer memory as it will be shown in 4.1.3. When desired, a spatial state can be easily derived from a state instance. For that, a unit $\mathbf{c}_a$ is *active*, when one of its subunits is *active*, i.e. when $U_a \cap U_{\text{act}} \neq \{\emptyset\}$ holds. Furthermore, a unit $\mathbf{c}_p$ is *predicted*, when one of its subunits is *predicted*, i.e. when $U_p \cap U_{\text{pred}} \neq \{\emptyset\}$ holds.

The way training in the TP is executed usually leads to only one *active* subunit per unit at one time step. Thus, when each unit has m subunits, the possible number of different U_{act} which instantiate a state S consisting of n *active* units, is m^n. A particular state can therefore be encoded with a lot of different state instances. During training, edges between subunits of those state instances are created. This allows for variable-order memory, which means that a prediction depends not only on the current state of a region, but also on past states.

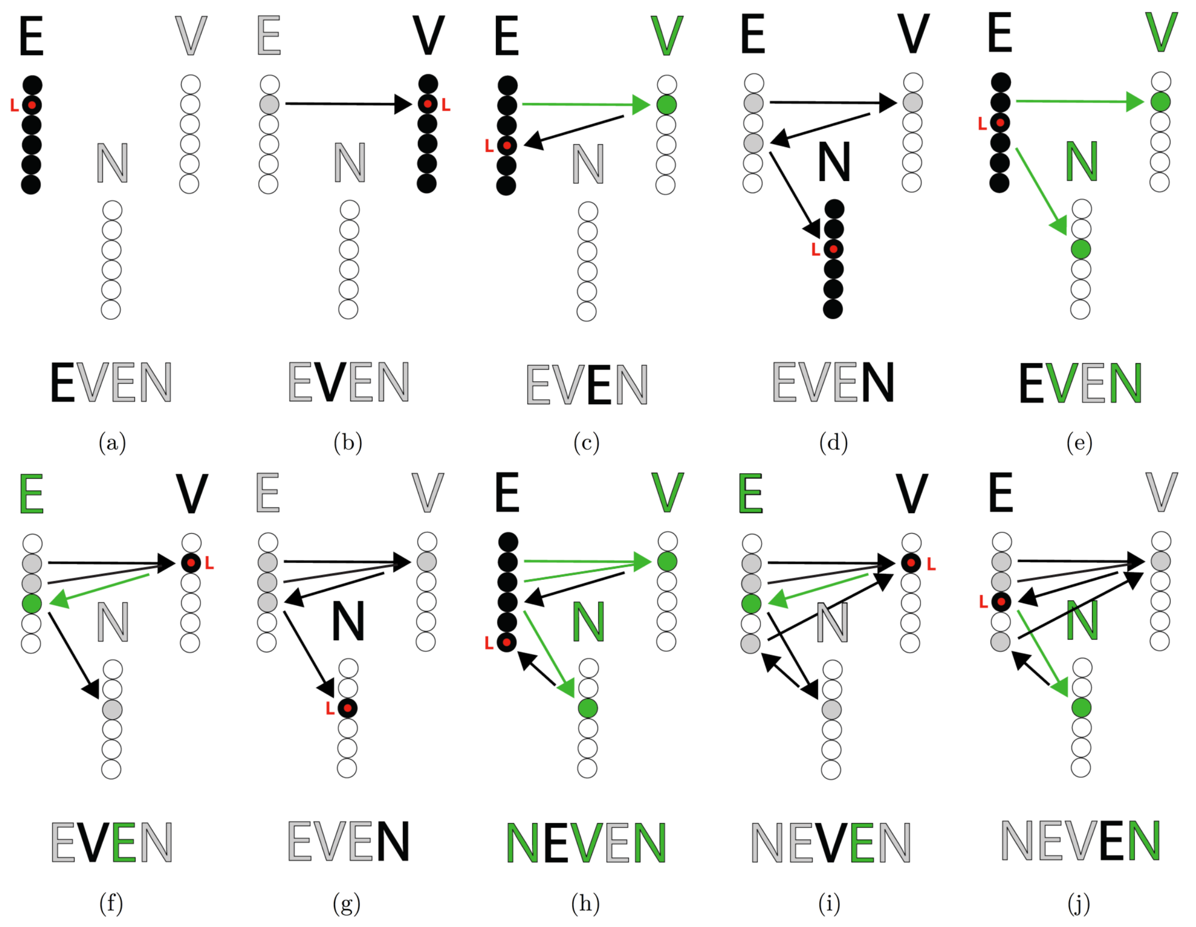


Figure 4.2: Training of the word ”EVEN”. Letters ”E”, ”V” and ”N” above the columns represent units, and circles represent subunits. A black subunit is in the *active* state, a green subunit is in the *predicted* state, and a unit with a red dot is in the *learning state*. Subunits which were already in the *learning state* since the training has started are colored in grey. The same color code applies to the letters ”E”, ”V” and ”N”. As each unit (i.e. each column) corresponds to a letter, those colors denote the states of the respective units.

States as well as instances of states are distributed over several units and subunits of a region, respectively. Therefore, it is not trivial to connect them forward in time. Another challenge is that in many problems a state S^t hardly ever reoccurs in the exact same way. Therefore, we need a mechanism to identify a past which is similar but not the same to what we have seen so far, when it comes to predicting the future. Let us assume we train a region on movies, due to the region's generalization ability the states of two frames of the training data might share 90% of their *active* units, but they are still dissimilar by 10%. However, for learning and recognizing sequences, it might be desired to look at those states as being identical. Those problems can be solved by using segments, as pointed out in the next subsection.

4.1.1.2 Segments

Segments bundle and separate sets of incoming connections to subunits. Since each subunit can create multiple segments during training, it can be used for different temporal paths. If one segment of a subunit is *active*, the subunit is considered to be *predicted*. A segment gets *active* if a specified ratio of its incoming edges is *active*[1]. That ratio has to be chosen empirically. In my implementation, it is set to 0.8.

Temporal training in an HTM region means to establish connections between state instances in order to learn to predict the next state instance given the current state instance. As stated above, a state instance is a set of subunits. Thus, in order to model a state transition, we need to connect two consecutive sets of subunits with edges. During training, a subunit which switches into the *learning state*, can create a segment which stores a list of edges from several subunits of the previously active set. A segment is not created, if the subunit was *predicted* in the preceding time step, because then there was already a segment causing this prediction. In this case, the list of incoming connections to the previously *active* segment can get extended to better fit the state instance which was active in the preceding time step. A segment does not have to connect to all subunits of the previous state instance. Usually, the set of subunits in a state instance is big enough to identify a specific state instance given only a part of its subunits. In addition, such a sparse connectivity increases robustness in recognizing a similar, but not identical past.

4.1.2 Initialization

In initialization, for each unit a fixed number of subunits is created. In contrast to the SP, no temporal edges are created from the outset. Instead, edges are created during training in order connect only those subunits which were *active* right after each other. The same is true for segments, which are created during training, too.

[1] An edge gets *active* if the subunit the edge is coming from is *active*.

4.1.3 Training

4.1.3.1 The States of Subunits

As the TP is trained online, it recognizes sequences and builds temporal connections at the same time. Therefore, a subunit's state determines, on the one hand, the state of the unit it is assigned to (i.e. recognition), and on the other hand, it determines the role of the subunit in training.

Active: A subunit can only be in the *active* state, if the unit it is assigned to is also in the *active* state due to feed-forward input. There are two ways that a subunit can switch to the *active* state. One way is that it was *predicted* in the previous time step and the unit it is assigned to actually got *active* due to feed-forward input in the current time step. The other possibility that a subunit gets *active* is, when a unit gets *active* due to feed-forward input but none of its subunits were *predicted* at the previous time step. In that case, all subunits assigned to that unit get *active*. An *active* subunit automatically activates all outgoing edges, which in turn may lead to predictions of other subunits.

Predicted: A *predicted* subunit automatically turns on the *predicted* state of the unit it is assigned to. A subunit gets *predicted* if one of its segments is *active*. This is the case if the ratio of *active* to *inactive* incoming edges to that segment exceeds a threshold. If a subunit was *predicted* in the previous time step and its unit actually gets *active* due to feed-forward input in the current time step, the subunit gets *active* in the current time step.

Learning state: Only subunits in the *learning state* create new segments and connections during training. A subunit switches into the *learning state* if it was *predicted* due to *active* subunits which were in the *learning state* (i.e. if the ratio of subunits in the *learning state* which *predicted* that subunit exceeds a threshold), and then got *active*, because the prediction was correct. Another way that a subunit can switch to the *learning state* is that it is the "best matching subunit" of all subunits of a unit which gets *active*. The best matching subunit has to be determined if no subunit of the currently *active* unit was *predicted* in the previous time step. The best matching subunit is the subunit with the segment which had the highest activation at the previous time step, even though that activation was not high enough to actually switch the segment to *active*.

4.1.3.2 A Training Example

The TP is designed to get trained in an online training process. A region is constantly predicting and learning at the same time. As we will see, prediction while training is not only a convenient feature of the TP, it is also necessary for determining a state instance at time step $t + 1$. In order to make the training process understandable, a simplified example will be used. The simplification results in a

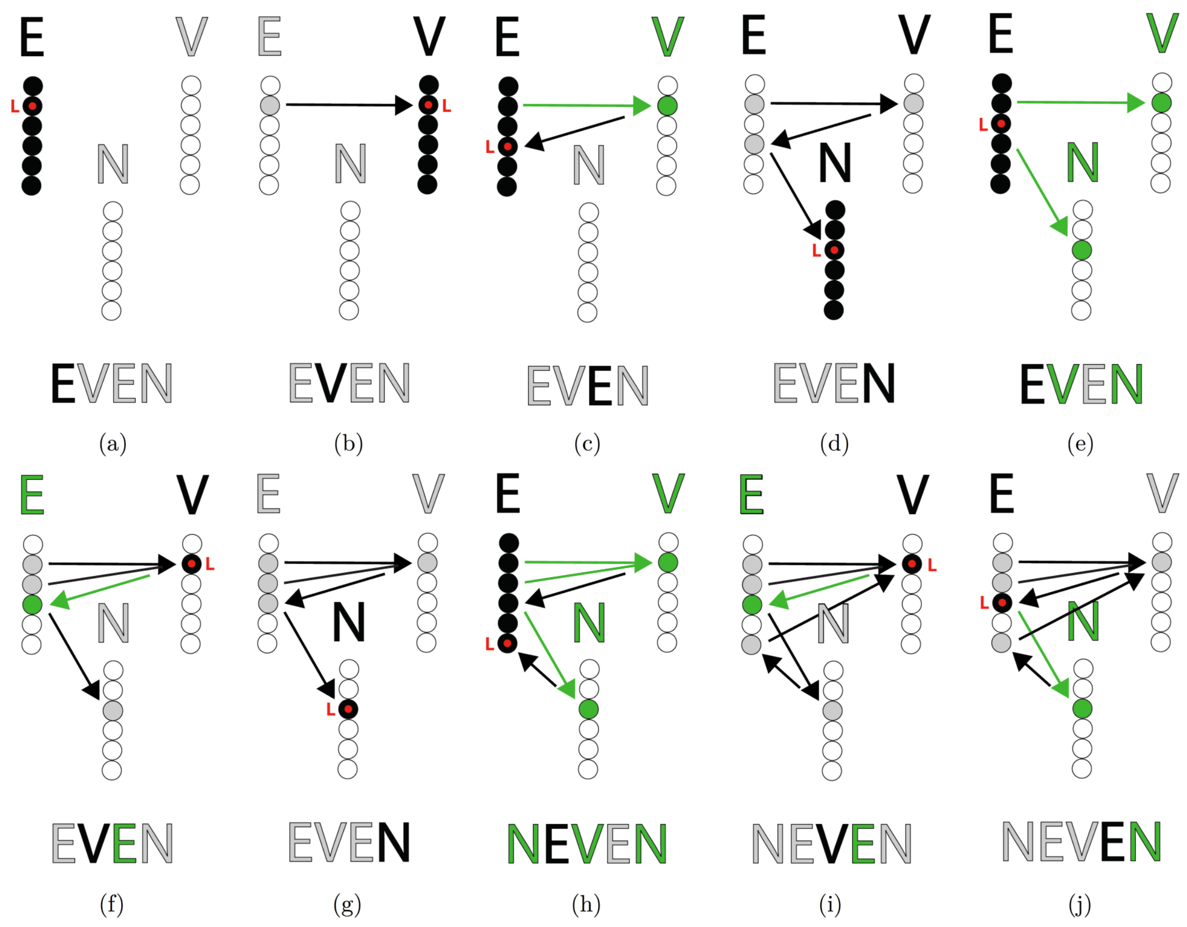


Figure 4.3: Training of the word "EVEN". Letters "E", "V" and "N" above the columns represent units, circles represent subunits. A black subunit is in the *active* state, a green subunit is in the *predicted* state, and grey means that this subunit was already in the *learning state*.

model, which connects sub-units without weighting the connections. This yields a second-order memory instead of a variable-order memory, proposed by Numenta. Reason for this simplification is that at the time this chapter was created, the respective information were not yet available. Nevertheless, that example is correct, assuming that all established connections have constant weights of 1 applied to them. It is also valid in that it correctly describes all possible states a region might be in during training, and what rules apply to each of them. Therefore, it helps to understand the principles of the TP.

We start with a region which is already trained with respect to SP (see Section 2.3.1). It consists of a set of units, where each has a column of subunits assigned to it. No temporal relationships are stored yet. Figure 4.3 shows a simplified illustration of a region, where segments are not shown. The region is spatially trained on the three letters ”E”, ”V” and ”N”. In this example, the state S of the region consists of only $n = 1$ *active* units.

(a) When the first input $\mathbf{x} =$ ”E” is observed, a state $S^t = c^1(\mathbf{x})$ is determined from the input where the ”E”-unit $\mathbf{c}_a$ receives the state *active*. Now, we have to decide for the state instance U_{act}. There are two possibilities for an *active* unit's subunits to get activated. If a unit is *active* and none of its subunits was *predicted* at the previous time step, all of its subunits get *active*. Why this makes sense will get clear at figure (c) and (e). If a unit is *active* and one or more of its subunits were *predicted*, only the previously *predicted* subunits get *active*. Following those rules, we can determine a set of *active* subunits at the current time step. In our example, none of the subunits was *predicted* because we have just started the sequence. Therefore, we activate the whole column of subunits. Then, we choose one *active* subunit per active column to be in the *learning state*. That subunit will instantiate its unit the next time this temporal context is observed. Chosen will be either the subunit with the least number of segments of all *active* subunits of a column, or in case of a prediction, the subunit which was *predicted* with the highest certainty. This is the subunit which is *predicted* due to a segment with the highest activation of all segments of all subunits of the unit (see 4.1.1.2). In our case, the second subunit was chosen at random, because there are neither any segments created yet, nor was any subunit *predicted*.

(b) The region observes a ”V”. Because none of its subunits was *predicted* in the previous time step, all subunits of the ”V”-column get activated. Again, a subunit is chosen at random for the *learning state*. Then, all subunits which were in the *learning state* at the previous time step forward-connect to all learning subunits at the current time step. For each transition, a segment has to be created which bundles all the incoming connections to a subunit. In our example, it would not be necessary to create a segment, because $n = 1$ and thus, there is only one edge for a transition to store.

(c) The region observes the second ”E”. Again, it was not *predicted*, therefore

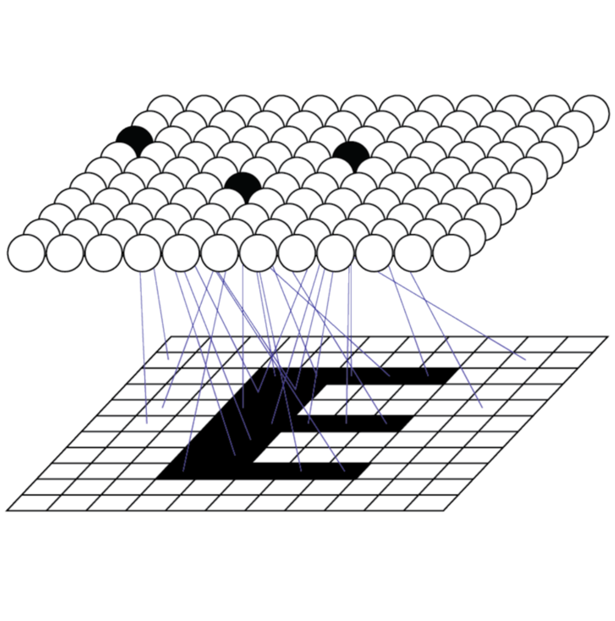


Figure 4.4: Illustration of how a TP connects subunits in a region with $n = 3$.

we activate all subunits of the ”E”-column. Every *active* subunit predicts through its outgoing connections. The second subunit of the ”E”-column has an outgoing connection to the second subunit of the ”V”-column, which switches to the *predicted* state. As in (a), we pick a random subunit for being in the *learning state*. We create a segment, assign it to the learning subunit and connect the previous learning subunits to the current one. From now on, this state instance, which consists of only one subunit, stands for ”E” after ”V” after ”E”. This makes it clear why forward-connecting state instances allows for higher order chains than just forward-connecting states.

(d) The region observes an ”N”. As in (b) and (c), a learning subunit is chosen at random and connected to the last learning subunit.

(e) The region has learned the sequence E-V-E-N. We restart presenting the sequence to the region. For that, all states of all subunits are reset. Otherwise a connection from the ”N”-column to the ”E”-column would be built. The ”E” is observed, which was not *predicted*. The whole column gets activated and all subunits of letters which occurred already after the ”E” switch into the *predicted* state.

(f) For the first time, an *active* unit’s subunit was *predicted* in the previous time step. Thus, only that subunit switches to both the *active* and the *learning* state. A new segment is created which bundles the incoming connection from the subunit with *learning state* before. That subunit has two segments now. Any *active* subunit predicts, therefore an ”E”-subunit is *predicted.* The region recognized the sequence and continues to compare actual states with *predicted* states.

(g) The ”E” was *predicted*, therefore it switched to *active* but no new edge or segment is created, because the subunit was *predicted* from subunits which were in the *learning state.* This is an important rule for the only case where no new segment is created. Then, the ”N” occurs which was *predicted* through an ”E”-subunit. The ”N” subunit switches to *active* and *learning* and again no new segment is created because the ”E”-subunit was in the *learning state.*

(h) We start over from the beginning of the sequence E-V-E-N but to demonstrate how the TP handles loops, we do not reset the states of the region. This results in the input sequence E-V-E-N-E-V-E-N. When the ”E” occurs, a connection to the ”N”-subunit is created. There was no prediction, therefore the whole column is active and predicts.

(i) The ”V” is observed and a connection to the last learning cell is created. From here, the region recognizes the sequence and as long as that sequence loops, no new learning takes place.

This example shows a few properties of the TP. In this example, it is a ”second-order” memory. As you can see, only one subunit of the ”V”-column was used throughout the whole learning process, even though the ”V” occurred twice after ”E” and once after ”E” after ”N”. The ”E”-subunit which was used to store the incoming connection from ”N” had information about what happened before, but that information was lost as the subunit of ”V” took over. However, Numenta speaks of its HTM as a variable order memory. Probably, I understood some details differently than Numenta. At the end of my work on this thesis, I could make use of the new mailing list of Numenta and got the answer that a smaller learning rate would lead to a higher-order memory. A smaller learning rate meant to not look at every newly created edge as already being connected, but to have a permanence value and a threshold similar to that in the SP, which slowly establishes potential connections.

Secondly, if there was no prediction in the previous time step, a whole column switches to the *active* state. That way we obtain the behavior of a first-order chain, which is extended to a higher-order chain as soon as the next state is known. That way, it is possible to recognize a sequence, even though it did not start from the beginning.

Figure 4.4 shows how the TP builds connections in regions with more than one *active* unit per state. Each *predicted* subunit has a segment, which bundles the 3 illustrated input connections. Every time the illustrated state instance of letter ”E” occurs, those segments get *active.* This leads to a prediction of its corresponding subunits and the combination of those is the *predicted* state instance of letter ”V”.

4.1.4 Problems with Temporal Pooling

In the introduced version of the TP, there are no transition probabilities between two states (i.e. the probability of state S^t given state S^{t-1} is not known). A transition property is binary, either there is a transition of a subunit u_k to a segment b_q ($u_k \rightarrow b_q$) or not. But there are problems where probabilities are of interest, like in the musical domain, which I tried to apply the HTM to (see Section 6.3). Some musical events occur very often in different temporal contexts. For example single notes should not just predict all possible next notes, but also their probability. However, the TP can be extended to a probabilistic model, as shown in 6.2.2.

4.1.4.1 Problems with Temporal Invariance

As pointed out in 2.3.2.1, if a region's feed-forward output is a union of *active* and *predicted* units, the spatial state of the parent region is a representation of those two states. Thus, states in higher regions get more stable with respect to time and predict several states forward in time, which seems to be a good concept at first sight. However, the major problem is that the parent region loses information about which order *active* and *predicted* units were in. Cases can be found where the actual order cannot be reconstructed any more. If the model should then generate sequences from what it learned, errors might occur. For a possible solution to this problem, see 6.2.1.

Another solution were to concatenate the vector of the current state S^k and the vector of the prediction S^k_{pred}, instead of merging them into a single vector by a pointwize OR operation. In that case, the reconstruction would naturally be separated and the order would be preserved. However, that would result in losing the temporal invariance property, because the input to the parent region between two time steps would not be similar any more and no stability would be gained in higher hierarchical levels. Maybe a mixed solution yielded the desired results which could be to concatenate $S^k_{\text{pred}} \cup S^k$ with S^k and S^k_{pred} and provide this as an input to the parent region. That way, at least some bits of the input remained stable over two time steps and it would still be possible to reconstruct S^k and S^k_{pred} separately. However, this has not yet been tried.

Another solution which has been tried, is that a region outputs a binary representation of active and predicted *state instances* instead of outputting active and predicted states. Intuitively, this would mean that the order of states would be preserved. However, in my experiments that method led to much higher complexity and sparsity for the SP of the parent region and did not yield the expected results.

5 Hierarchical Topology

In the previous sections we described SP based on a region which clusters input data in an unsupervised way. A trained region holds a set of cluster centers (QCs), which are distributed over the input space. If an input vector is exposed to the region, a fixed number of closest QCs are found. This QC subset is then called the region's state, which tends to be similar for similar input vectors. Such a procedure allows for a quite powerful modeling of statistical dependencies over the input space.

However, the actual strength of an HTM comes from its hierarchical topology, built from multiple regions. A child region's state gets transformed into an output vector, which in turn serves as an input for its parent region. Not only does data get compressed further by this upward-pass, which we will refer to as feed-forward process. A region which has multiple child regions can also model, for example, dependencies between multiple sensors. Learning such dependencies directly from uncompressed data might lead to unmanageable processing effort. Learning dependencies at higher hierarchical levels, however, results in an acceptable complexity.

5.1 Topologies

In [Hawkins and George, 2006] it is pointed out that concatenated input vectors should be as correlated as possible, which is reasonable in that uncorrelated input to a region would result in many different clusters to learn. Let us consider lossless learning for a thought experiment. If we trained a region on purely random binary input data, all possible bit combinations would have to be stored. For an input vector of size N, this would result in 2^N archetypes. Therefore, even if a region is provided with correlated input data, the necessary size M of the region might exceed the size N of the input. Depending on how lossy we allow the data compression to be, the size of a region can be varied, but the the necessary size of a region tends to increase, if the input is less correlated. To minimize the complexity of storing combinations of input bits, Numenta partitioned images into several small areas and combined close areas in regions of lower hierarchical levels and more distant areas by continued combination of regions while moving up the hierarchy. Numenta opted for that topology because of the assumption that close pixels in images are more strongly correlated than those further apart.

To minimize the ratio (memory space requirement / quality of reconstruction), for most problems, the optimal number of child regions to a parent region is two, because the more children, the more uncorrelated the data. Information from multiple inputs can still be combined in higher hierarchy levels.

Additionally, it can make sense for an input to have multiple parents. If inputs X and Y are conditionally independent given input Z, there is no need for combining X and Y in any hierarchical level. Instead, Z has two parent regions. One region combines X and Z, the other region combines Y and Z.

5.2 Quantization Center Spaces

A QC space is the input space of a parent region, created by simply concatenating output vectors of its child regions. Furthermore, an output vector of a region is the binary representation of the region's current state S^t. On-bits in the output vector correspond to *active* units in the region, and off-bits represent *inactive* units (see Figure 2.2). To enforce temporal invariance in higher regions, also *predicted* units are represented as on-bits in the same output vector. That way, the output vector is self-similar over two time steps and so is the representation of the parent region. A variant of that can be that *active* units at time $t-1$ are represented as on-bits, too.

For the Sparse Distributed Representations for Continuous Input (SDRC) method, which will be introduced later (see Section 7.2.1), the output of a region might again be real-valued. To that end, instead of an on-bit, for each *active* unit a value can be set, which is proportional to the closeness of that unit to the region's current input vector. This increases the precision for further processing in higher hierarchical levels.

5.3 Feed-Forward

In order to recursively compute the region's states in hierarchical topologies, the states of all regions are computed iteratively from the bottom to the top, where the input to a parent region is derived from the state of its child region as described in Section 5.2.

5.4 Prediction

After feed-forward was executed, each region predicts possible next states as described in Section 4.1.3 and Section 6.2.2. Those predictions can then be combined via feedback in order to receive better overall predictions by utilizing information of all regions. To that end, the sets of *predicted* units of lower regions in the hierarchy are reduced by intersecting each set with reconstructions of predictions of higher regions. The approach to feedback which was used in my implementation is described in detail in Section 6.2.3.

5.5 Training

Spatial training can be done iteratively by using the feed-forward procedure only. We start to train all regions in the lowest hierarchical levels. Then, the training for the next level can be started and so on. This approach ensures the fastest convergence of the whole model.

Training can theoretically be done simultaneously over all hierarchical levels, too. In this case, a slower convergence of regions at higher hierarchical levels is to be expected. In Chapter 7 a few ideas for an improvement of the SP are introduced. One idea is to compute the state of a region in training by using information from both feed-forward and feedback. In such a case, simultaneous training of all regions of the model would be necessary (see Section 7.4.1).

6 Implementation and Results

6.1 Introduction

The goal of my experiment was to implement the HTM in a Java program in order to examine its ability to learn and generate musical material. The expectation was that the HTM can be trained on sequences of different musical event types (e.g. notes, chords, and scales), where it should learn several correlations. One expected achievement was that the HTM learns to represent simple sequences of states within regions. Another expectation was that those sequences could be transformed into temporally invariant spatial representations in parent regions. Then, parent regions would model transitions of those representations. Finally, I expected regions on top of the hierarchy to learn correlations between those abstract, temporally invariant representations of sequences of different musical event types. From that model, I wanted to generate new musical material by feeding back and combining the predictions of all regions.

Temporal invariant representations in the musical domain can have the benefit that they enable the model to learn long-term dependencies of musical events. Representations in higher regions of a HTM topology which change more slowly over time, generally represent longer timespans than representations in lower regions do. Therefore, they could learn features which can only be derived by having a view on longer timespans. Properties like the current scale of a song, its genre, the current chord or a whole cadence could then emerge as a representation. By making decisions on a more granular level depending on such high-level features, a better generative model could be built.

However, it turned out that some of the underlying principles of the HTM are difficult to realize in practice, even though they seem to be simple and logical in theory. One of the first and most obvious problems was that an HTM wastes a lot of subunits if the input does not change over several time steps. A region creates an explicit state instance for each time step to store sequences of states, even if the state of the region does not change. To overcome this problem, I allowed connections of a state instance to itself (see Section 6.2.4). By doing so, only the order of events was preserved, while information about when a certain event occurs was lost. To solve this problem, for each event type an additional input was created, which presents the onset times of the corresponding event type to the HTM (see Section 6.2.5).

Having solved this problem, I found out that temporal invariance is not difficult to achieve, but it is difficult to make use of it for predicting the next state, especially when the HTM should be used as a generative model (see Section 4.1.4.1). Thus, I did not include *predicted* units to the region's outputs so that temporal invariance could not emerge (see Section 5.2). I rather concentrated on the HTMs ability to store and recognize long sequences of states within a region. This was done by modifying the original training algorithm to increase the order of the temporal memory (see Section 6.2.1). In addition, I tried to provide enough timing information as input, so that the HTM can reason from that which stabilizes the model at generation (see Section 6.3.1).

Another problem was to find the right topology for the experiment. The most intuitive setup was that each event type plus its onset time is concatenated and fed into a region at a quite low hierarchical level. Thus, a representation in such a region would mean, for example, "a certain note at a certain time". However, representations of concatenated inputs are best learnt if the inputs are highly correlated. That is not so much the case for pitches and their onset times than it is for chords and pitches. This led to a topology where events and their onset times are combined in a quite loose way, while notes and chords are combined more tightly (see Section 6.3.1).

The biggest overall issue was that the original HTM is not designed to be a generative model. Even though it sounds logical that a good predictor is supposed to be a good generator, there are some missing mechanisms. A general problem is that predictions have to be really reliable because within a few steps from the start of a generation process, all knowledge about the past and the current context is based exclusively on past predictions. For example, it can happen that while the model generates data, it falls into an inconsistent state. When low-level regions of two different event types predict several next states each, combinations of such predictions can be found which never occurred during training. If such inconsistent states are chosen as next states, the parent region may stop to predict. This problem is, on the one hand, a sign that the training data is too sparse, on the other hand it is a sign that the model has overfitted. However, it was not possible to find good parameters to amend that problem. If I tried to train the model more slowly, if I decreased the number of units, or if I reduced the threshold for predictions, the quality of the generated data decreased rapidly. Finally, that problem was solved by a back and forth procedure, which prevents inconsistent event combinations during generation (see Section 6.3.2).

A reason for the before-mentioned problem might be that both vertical and horizontal connections have no real-valued weights assigned. As soon as a connection is established, it is considered to be fully connected. The decision if a unit or a subunit is *active* or *predicted* is solely based on the number of established input connections, and the *active* or *predicted* states are binary, too. This does neither allow for a good generalization nor for a smooth modeling of the feature space, and overfitting happens easily. The only way to smoother representations and temporal transitions were to increase the number of units and subunits, respectively. This in turn leads to unnecessary high memory consumption and processing effort.

6.2 Modifications to the Original Model

6.2.1 Modified Training

As pointed out in Section 4.1.3, the TP as I implemented it, has a temporal memory of only two steps in the past. Therefore, I was forced to implement a rough workaround in order to make the model create meaningful data at all. The modified TP was only used for the lowest regions of the hierarchy, as they were in charge of building long temporal sequences, while higher regions were responsible for providing context information. The reason that I did not apply the modified TP to all regions is that it solely works for states with only one *active* unit ($n = 1$), as described in the end of this section.

In the original training, if a state at time t was not *predicted* at time $t - 1$, a whole column of subunits becomes *active* and the order is decreased to 1. In combination with the rule that no new connections are created when a unit was *predicted*, this results in an order of 2. To understand both the original concept and the description of the modified TP, it is recommended to look at Section 4.1.3.

The basic idea of the modified training is that there are two modes of the TP. One mode is the training mode, which works slightly differently to the original training. The other is the prediction mode, which does not differ from the original TP. In training mode, the original TP was modified such that the order is never decreased, independent of whether the current state was *predicted* or not. This can be basically achieved by never activating a whole column during training. Instead, only one subunit per column is chosen to be *active*, as described below.

In the following it will be shown how a region learns sequences in the modified version of the training. As in the description of the original training, we start with a region which is already trained with respect to the SP. It consists of a set of units, where each has a column of subunits assigned to it. No temporal relationships are stored yet. Figure 6.1 shows a simplified illustration of such a region which is spatially trained on the three letters ”E”, ”V” and ”N”. The sequence E-V-E-N is input repeatedly in order to show how the region is able to handle already observed sequences. In this example, the state S of the region consists of only $n = 1$ *active* units.

(a) The region observes an ”E” and activates the ”E”-unit, which was not *predicted*. In contrast to the original training, where in such a case all subunits of the ”E”-column would become *active*, we activate only the learning cell. The way how

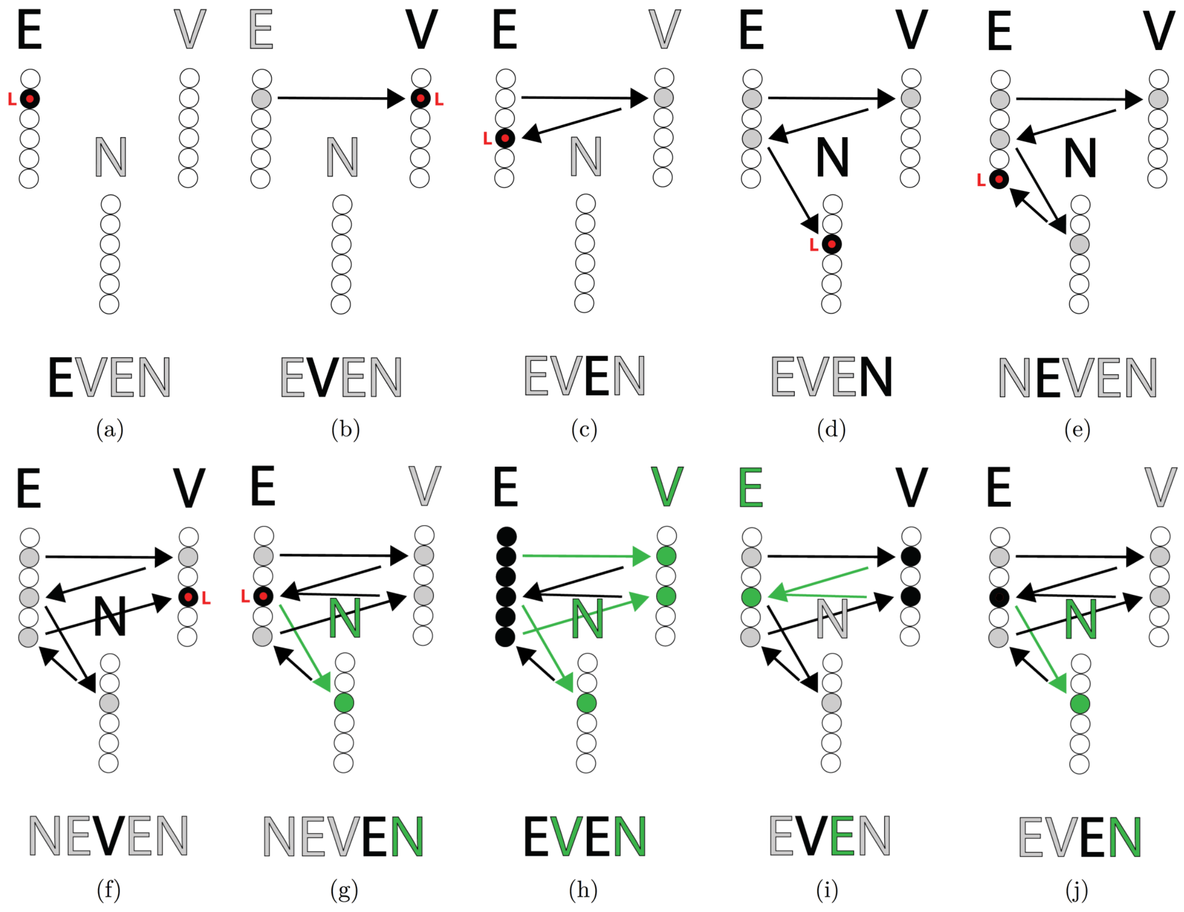


Figure 6.1: Modified training of the TP for the word ”EVEN”.

the learning cell is found remains the same as in the original training, which in this case is simply at random.

(b) The ”V” is observed. Again, a learning cell gets picked at random and it switches to *active*. A segment and a corresponding connection is created, which points from the last learning cell to the current learning cell.

In (c), (d), (e), and (f), the progress of learning for the repeated sequence E-V-E-N is shown, as described in (a) and (b). As we can see, the sequence, even though it starts from the beginning, is not recognized during the learning mode. Thus, a redundant parallel path gets created. This can be seen as maximizing the order, because it can make sense to store repetitions of a sequence, too.

(g) Eventually a subunit has to be reused, so that there is a chance to stop the redundant path from growing. I solved this by introducing a probability for reusing an already used subunit. However, there are no rules on which of the already used subunits to choose. If a subunit is chosen which predicts the next state correctly, the subunit was reused without an error. However, this is not very likely, especially if many subunits are assigned to each unit. In most cases, the subunit is already responsible for another path. If a subunit was made responsible for two paths during training, it predicts erroneously in prediction mode. Such errors have been reduced to a minimum by using the knowledge of parent regions which have a better knowledge of the context, because they receive input from multiple regions (see Section 6.3.1).

(h) Now we assume that the training has finished and the TP is set to prediction mode. In that mode, the TP works identical to the original TP. For each unit which was not *predicted* at the previous time step, a whole column becomes *active* and all of its subunits predict the next states.

(i) The *predicted* unit actually occurred, the *predicted* subunits switch to *active* and predict the next subunit. That subunit was reused for the sequence and joins the two paths into one.

(j) This is the end of the first repetition of the sequence E-V-E-N. If it started anew, it would only use one of the two redundant paths, which is the one representing a repetition.

Note that this method does not lead to any improvement for states with more than one *active* unit ($n > 1$). Reusing a set of random subunits is similar to picking a random state instance given a state. Due to the vast amount of different possible state instances as n increases (see Section 4.1.1.1), it is very unlikely to choose exactly that state instance which correctly predicts the next state. But this would be necessary to ”close” a path, so that not every repetition is stored as a new sequence. For my purpose it was not necessary to have states of $n > 1$ in regions in the lowest hierarchical level, because they simply projected a one-out-of-n input representation onto one-out-of-m units in the region (see Section 6.3.1) and were only responsible for building high-order chains.

6.2.2 Probabilistic Prediction

The goal of probabilistic prediction is to compute $p(\mathbf{c}_j^t \mid S^{t-1})$ of a region. Based on the training of the TP, ”horizontal” connections within a region are built and we know which units are *predicted* following the previous state. However, the *predicted* property is binary, either a unit is *predicted* or not. The probability of a unit given the previous state can be basically derived by counting how often a *predicted* subunit actually became *active* during training. To that end, we simply store that count for all edges. Let $o(\mathbf{c}_j^{t-1} \to b_q) \in \mathbb{N}$ be the count of correct predictions of segment b_q by an outgoing edge of a unit $\mathbf{c}_j$ which was *active* at time $t-1$. A prediction is correct if the corresponding subunit of an *active* segment b_q actually became *active* at time step t. If there is no edge $\mathbf{c}_j^{t-1} \to b_q$, then $o(\mathbf{c}_j^{t-1} \to b_q) = 0$.

The probability of a segment b_q given an *active* unit $\mathbf{c}_j$ at time $t-1$ is

$$p(b_q \mid \mathbf{c}_j^{t-1}) = \frac{o(\mathbf{c}_j^{t-1} \to b_q)}{(\sum_{m=1}^{M} o(\mathbf{c}_j^{t-1} \to b_m))/n}, \tag{6.1}$$

where M is the total count of segments in the region and n is the number of *active* units per state S.

The probability of a segment b_q given all *active* units $\mathbf{c}_a$ at time $t-1$ is

$$p(b_q \mid \mathbf{c}_a^{t-1}) = \frac{\sum_a p(b_q \mid \mathbf{c}_a^{t-1})}{\sum_a H(o(\mathbf{c}_a^{t-1} \to b_q) - 1)}, \tag{6.2}$$

where $H(\cdot)$ is the Heaviside function, which yields 1 for a positive argument and 0 for a negative argument. The probability of a subunit given $\mathbf{c}_a^{t-1}$ is the average probability of all of its *active* segments given $\mathbf{c}_a^{t-1}$. Finally, the probability of a unit given the previous state, $p(\mathbf{c}_j^t \mid S^{t-1})$, is the average probability of all of its subunits which are *predicted* given S^{t-1}, where $S^{t-1} = \mathbf{c}_a^{t-1}$.

6.2.3 Prediction-Feedback with Intersection

The term ”Intersection” originates in a setup where both the reconstruction and the prediction of a unit is binary, as it is the case in the description of [Hawkins, 2011]. Prediction-feedback from a parent region to its child regions is the reconstruction of the prediction of a parent region. Intersection means that the set of units in a child region which are *feedback-predicted* is intersected with the set of *predicted* units of the child region. The resulting set of units is then used for prediction-feedback to the child of the child region, where the reconstruction is again intersected with a prediction, and so on.

For reconstruction of a prediction of an input bit $x_i = \mathbf{c}_i^0$, the following recursive

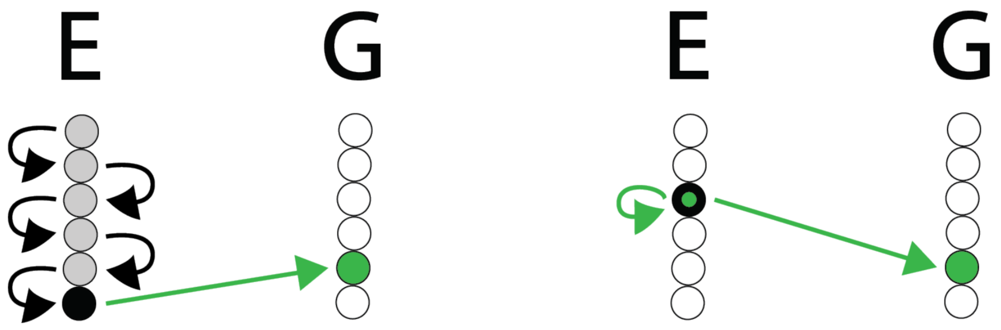


Figure 6.2: Left: If the input to a region remains unchanged over several time steps, subunits of *active* units are wasted and the previous state gets forgotten. **Right:** Subunits connect to themselves to avoid that behavior.

method was used in my implementation.

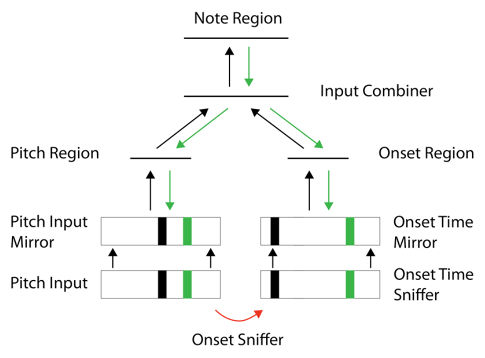
$$r(\mathbf{c}_i^k \mid S_{\text{pred}}^{k+1}, \dots, S_{\text{pred}}^G) = \frac{\sum_j p(\mathbf{c}_j^{k+1} \mid S_{t-1}^{k+1}) r(\mathbf{c}_j^{k+1} \mid S_{\text{pred}}^{k+2}, \dots, S_{\text{pred}}^G) w_{ij}}{\sum_i \sum_j r(\mathbf{c}_j^{k+1} \mid S_{\text{pred}}^{k+2}, \dots, S_{\text{pred}}^G) w_{ij}}, \tag{6.3}$$

where S_{pred}^k is the set of all *predicted* units in a region in the k-th hierarchy level $k = \{1, \dots, G)$ and $r(\mathbf{c}_j \mid S^{G+1}) = 1$. S_{t-1}^k is the previous state of a region in the k-th hierarchy level and for all variables without time specification, the current time t is to be assumed. $p(\mathbf{c}_j^{k+1} \mid S_{t-1}^{k+1})$ is the probability of a unit given the previous state, as described in Section 6.2.2. Based on the resulting likelihood function over all $\mathbf{c}_i^0$, probabilistic sampling can be performed as described in Section 6.3.2.

6.2.4 Self-prediction of Subunits

In the original training algorithm, at each time step a new state instance is determined and connected to the previous state instance (see Section 4.1.3). With this method it is basically possible to represent states which last for longer than one time step, while preserving information about their duration. However, if a state lasts for longer than the order of the chain, the previous state gets forgotten. The implemented solution to this problem is that if a state does not change during two time steps, the state instance remains the same, too. That way, units predict themselves as illustrated in Figure 6.2.

By doing so, information about the duration of every state gets lost. A state of a spatially trained region does not change as long as the region's input does not change. Therefore, if another input to the HTM provides the duration of the current event, all necessary information is available again. As described in the



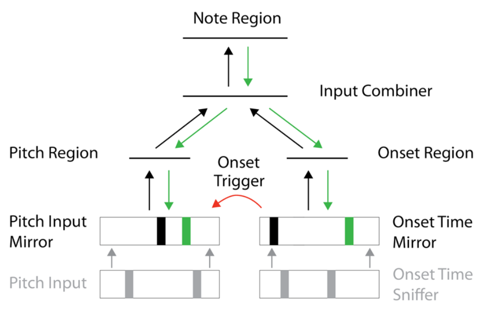


Figure 6.3: A black rectangle represents a current on-bit in the input vector, a green rectangle represents a predicted on-bit. **Left:** During training, the *Onset Time Sniffer* reflects any change in the *Pitch Input* in that it provides the time of a change as a binary vector. **Right:** During generation, if the actual time is equal to the predicted time, the *Onset Trigger* causes the *Pitch Input* to realize its prediction.

next section, I decided not to provide the duration of an event, but the time of occurrence relative to the measure, so that stored onset times are more directly tied to the meter.

6.2.5 Representing Time

As described above, for representing events of different durations, it is necessary to provide sequences of events and sequences of their durations as separate inputs. Figure 6.3 illustrates a simple setup of an HTM with two inputs, two "mirrors", three regions, and an input combiner. The provided event types in this example are *pitch* and *pitch onset time*.

The *pitch* input is implemented as a module which reads data directly from the training corpus. In contrast, the *pitch onset time* input data is created during training. This "Onset Sniffer" monitors the pitch input. It reacts on any changes in the monitored input in that it represents the time of a change as a binary vector. In that vector, all bits are off except the bit which represents the relevant time. In my implementation, that time is always provided relative to the current bar.

Between the actual inputs and the HTM there are input mirrors, which forward the input from below during training and mirror the predictions of the HTM during generation. In generation, mirrors sample from the feedback distribution of predicted next states (see Section 6.3.2) and provide the result as an input to the HTM (see Figure 6.3). In order to synchronize several event types with varying onset times, each event type mirror has a corresponding Onset Trigger mirror, which

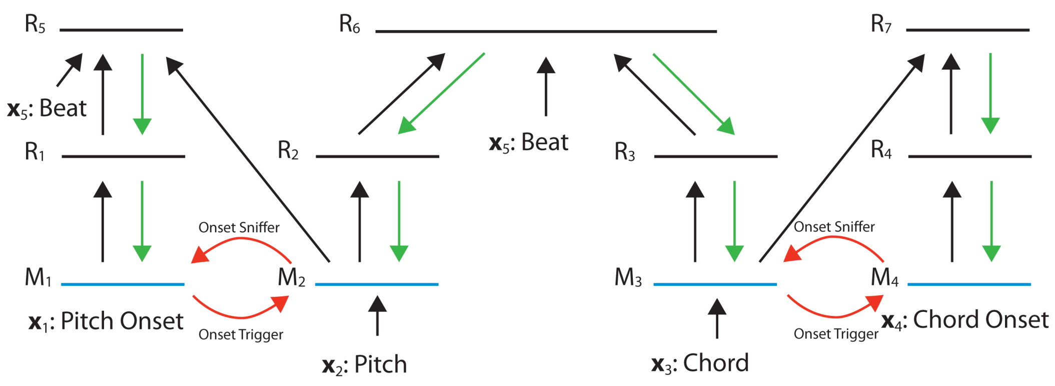


Figure 6.4: Experiment setup.

takes care of switching to the sampled prediction at the correct time. As soon as the time in the measure equals the sampled predicted onset time of the Onset Trigger, it activates its slave and both mirrors switch to their sampled predictions.

6.3 Implementation

6.3.1 Setup

My setup consists of five inputs, seven regions and four mirrors (see Figure 6.4). The main input are sequences of pitches and chords. A mirror forwards the input from below as long as the HTM learns, but for generation, it mirrors the predictions of the HTM (see Section 6.2.5). During training, mirror M_1 is an ”onset sniffer”, which monitors mirror M_2. Every time the input to mirror M_2 changes, M_1 reflects that change by outputting the current time relative to the current bar. The same is true for mirror M_3 and mirror M_4. This is a kind of a derived input, which accounts for the problem that the HTM is not able to efficiently represent durations of events (see Section 6.2.4). The actual output of onset mirrors $M_1 = \mathbf{x}_1$ and $M_4 = \mathbf{x}_4$ is a binary vector of length 16. Such a vector represents a measure, therefore a measure has a resolution of 16. An onset time is represented as a single on-bit in that vector. For example, in four-four time an onset at the second beat would be represented as an on-bit at the fifth position of the vector, while all other bits are off.

During the generation process, an ”onset sniffer” turns into an ”onset trigger”. It receives feedback from the HTM, which is a vector of predicted probabilities for the next onset time (see Section 6.2.2). The trigger samples the next onset time from this distribution and triggers its corresponding event mirror, which in turn samples from the distribution it receives through feedback. Then, both mirrors output their sampled result.

Input $\mathbf{x}_2$ is responsible for pitch. This is a binary vector of length 56, which is the overall pitch range of the training corpus. A pitch is simply represented as a single on-bit in the vector, relative to the lowest pitch in the pitch range. Note that all the training data used in this experiment is monophonic. Therefore there will be never more than one on-bit in vector $\mathbf{x}_2$.

Vector $\mathbf{x}_3$ is of length 24 and inputs the current chord. Again, each possible chord has a corresponding bit in vector $\mathbf{x}_3$. There are two possible chord types, major and minor, and for each chord type there are 12 halftones for the key note of the chord. Therefore, there are 24 different chords to represent.

Input $\mathbf{x}_5$ is a binary vector of length 4 and represents the current beat of a measure. Since all chosen songs are in 4/4 time, that representation is sufficient. This vector is still fed into the HTM during generation and helps stabilizing the output in the generation process. A basic rule can be stated: If it cannot be guaranteed that the order of a chain is high enough for a certain purpose, try to make the states of the region more diverse. For example, if a long input sequence of two alternating events were fed into a region, the region would learn two alternating states from those events. With an order of two, the region would be in the exact same state after one repetition of those alternating events. However, if an additional input which counts the number of repetitions were concatenated to the original input, the state of the region would be different for every repetition. That way, an actual order of 2 could be artificially extended to an arbitrary order by utilizing the memory of the SP. This property is exploited by the input $\mathbf{x}_5$.

Regions R_1, R_2, R_3 and R_4 share similar properties. All of them receive a very simple input, which is a binary vector with a single on-bit. Therefore, it is sufficient that the number of *active* units per state is $n = 1$. Here, the only goal of the SP is to assign a unit to each input dimension. Due to the simplicity of the training data, the SP used is the original SDR pooler introduced in [Hawkins, 2011], (see Section 3.3.2). Regions R_1 and R_2 have 100 units and there are 20 subunits assigned to each unit. Regions R_3 and R_4 have 50 units and there are 5 subunits assigned to each unit.

For the number of edges per region which connect units to input bits, I used a measure denoted by *synapse coverage* η. That value specifies the fraction of the number of edges if every unit of a region were fully connected to the input. Hence, the number of input connections to a region is $\eta * |C| * |x|$, where C is the set of units of the region. Edges are created in initialization, so that each edge connects a random unit to a random input dimension. All regions of the first hierarchy level have a *synapse coverage* of 0.005.

The regions of the first hierarchy level have the only purpose of constructing chains with the highest order possible. They activate only one unit per state and are trained by the modified TP training introduced in 6.2.1. All other regions are trained with the original TP training.

Region R_6 is responsible for learning correlations of chords with single notes of a melody. Because the current position in a measure can matter in deciding for the next pitch and chord, beat vector $\mathbf{x}_5$ is used as an additional input to the region. As the beat vector is of size 4, it provides only a rough position in the measure. A higher resolution would increase the sparsity of the representations which would lead to less degrees of freedom in generation. Region R_6 has a total number of 600 units and the number n of *active* units per state is 10.

Region R_5 is responsible for the rhythm of the melody. Here, the most important additional information is the pitch that the current onset belongs to. For instance, the model could learn that the first note on the first beat in the measure might have a longer duration if the pitch is tonal, than if the pitch is dissonant. Due to the fact that all training instances are normalized with respect to the key, the pitch can have some implicit information about tonality. But in general, if a melody should be recalled, rhythm and pitch are of equal importance. Input $\mathbf{x}_5$ is again used to stabilize the output at generation. As a unit count for region R_5, 600 was chosen, with 1 subunits each. The number n of *active* units per state is 10.

Region R_7 is very similar to R_6 with the difference that the input it has to model is much simpler. The variance of onset times of chords is smaller than that of notes and also the number of different possible chords is smaller. Therefore, the unit count is set to 50 and the amount of subunits per unit is 5 and $n = 1$.

Note that the input is encoded in a one-out-of-n representation. Otherwise, it would be difficult to decide for the "right" combination of bits, when there are several combinations predicted. This problem could be solved by sampling a single solution from the prediction (see Section 7.2.7).

6.3.2 Generating Music

The generation starts by switching from training mode to generation mode without resetting the model. That way, the states and the predicted states of all regions are consistent. In the generation mode, the following loop is processed. The notation of variables corresponds to the notation in Figure 6.4.

1. Compute feedback with intersection (see Section 6.2.3) and reconstruct the resulting predictions.

2. Consider each reconstruction as a probability distribution and sample from it by repeatedly picking one bit at random and choosing it with its corresponding probability, until any bit is chosen.

3. Find the onset trigger M_1 or M_4, whose sampled onset time is closest to the current time t, set t to that onset time.

4. Realize[1] the prediction in the mirror found in 3. Realize the prediction in its corresponding trigger slave mirror.
5. Update the states of all regions based on the new inputs.
6. Compute predictions, based on the new states.
7. Goto 1.

If the sampled onset time of both onset trigger in step 3 is equal, instead of step 4, the following steps are processed. Without that method, sampling takes place based on predictions given only the past. However, if more than one event's prediction should occur simultaneously, consistency has to be ensured by sampling one event after the other.

1. Realize the onset prediction of $\mathbf{x}_1$ and $\mathbf{x}_3$, and the event prediction of $\mathbf{x}_4$.
2. Set all *predicted* units of region R_3 to *not predicted*, when they have no connection to the realized prediction of $\mathbf{x}_3$.
3. Set all *predicted* units of region R_6 to *not predicted*, when they have no connection to the residual *predicted* units of R_3.
4. Compute feedback with intersection to retrieve a prediction in $\mathbf{x}_2$.
5. Sample and realize the pitch prediction of $\mathbf{x}_2$.

By doing so, only those predictions of pitch in M_2 and chord in M_3 remain, which have common connected units in R_6. That way, only those notes are allowed which already occurred together with the sampled chord.

6.4 Results

6.4.1 Quantitative Evaluation

In order to evaluate the HTM setup described in Section 6.3.1, three different methods are used. In Section 6.4.1.1, we compare standard deviations of pitch and duration of melody notes between training data and generated data. In Section 6.4.1.2, results of an evaluation via cross-entropy are presented, and Section 6.4.1.3 shows histograms of the length of identical sequences in training data and generated data.

[1]Realizing a prediction means that the input $\mathbf{x}$ of an event type is set to what was sampled from the prediction of this event type. This results in an $\mathbf{x}$ where all bits are off, except the sampled position, which is on.

Song	Notes	Chords	Duration	Tact	Scale
A Hard Days Night	305	91	336	4/4	major
California Dreamin	157	90	304	4/4	major
Dock of the Bay	271	62	256	4/4	major
Hey Jude	530	107	528	4/4	major
House of the Rising Sun	190	164	688	4/4	major
Like a Rolling Stone	495	189	576	4/4	major
Losing my Religion	319	76	560	4/4	major
Mr. Tambourine Man	622	205	944	4/4	major
Norwegian Wood	133	33	240	4/4	major
Nothing Compares 2 You	330	89	304	4/4	major
No Woman No Cry	349	168	336	4/4	major
Paint it Black	450	100	576	4/4	major
Ring of Fire	222	90	288	4/4	major
Sounds of Silence	282	61	336	4/4	major
The Boxer	517	109	480	4/4	major
When a Man Loves a Women	285	80	176	4/4	major
While my Guitar Gently Weeps	178	115	544	4/4	major
With or Without You	249	116	528	4/4	major
Yesterday	179	68	192	4/4	major
Your Song	387	99	512	4/4	major

Table 6.1: The nr. of notes, the nr. of chord changes, the duration in beats, the tact and the mode for songs used for training and evaluation of the respective models.

Evaluation is carried out on 20 single pieces of the Temperley Rock Corpus (TRC), which includes monophonic melodies and chords of the 100 top-ranked songs of the Rolling Stone magazine's list of the "500 Greatest Songs of All Time" [De Clercq and Temperley, 2011]. Before training, each song is transposed to a default key. In order to decrease the number of possible chords, all chords are simplified to either major or minor. For a list of the songs used, see Table 6.1.

The model parameters described in Section 6.3.1 are empirically optimized in order to produce the best possible outcomes with respect to the quality of sampled music. We call that original model "ORIG". In order to evaluate at which point the quality of the sampled music decreases when some of the model parameters have changed, two different setups are created. In one of the "degenerated" models, the size of region R_6 is reduced from a size of 600 units to a size of 25 units, and n is reduced from 10 to 3 (see Figure 6.4). Therefore, we will refer to this setup as "SMALLR_6". In the other "degenerated" model, the size of region R_5 is reduced from 600 to 25 units and again n is reduced from 10 to 3. Therefore, we will call it "SMALLR_5". In the original model, the size of units was chosen so high, because it was optimized to be trained on the whole corpus in the first place.

In this evaluation chapter, however, the HTM is trained on each of the 20 songs separately, and a reduction to 25 units is found to yield results, which are slightly worse than those of the original model. From all of those 20 trained models, a song with the same number of notes as the original song is generated. Then, the original

	r		p-value	
Model	Pitch	Duration	Pitch	Duration
ORIG	**0.8905**	**0.8943**	**1.4339E-7**	**1.0634E-7**
SMALLR_6	0.6889	0.5968	7.8164E-4	5.4703E-3
SMALLR_5	0.8419	0.7226	3.2558E-6	3.1996E-4

Table 6.2: Three different model architectures were trained on 20 songs each, one song at a time. For each learned song, an output was generated and the Standard Deviation (SD) of pitch and duration of the original song and that of the generated counterpart was calculated. Subsequently, for each model, the correlation coefficient r of the original song's SDs and the sampled song's SDs, and the corresponding p-values were calculated. The results show that the model is able to generate music with SDs of notes and note durations similar to those learned from the data. The correlations are higher in the original model, than those of the models with reduced layer sizes. The p-values < 0.01 show that those results are statistically significant. Note that the model SMALLR_5 has a reduced layer responsible for modeling note onset times, therefore, the correlation of note durations deteriorates more than that of pitch. In SMALLR_6, the main layer's size is reduced, which has indirect influence on the layer R_5, because there is a connection from Mirror M_2 to R_5 (see 6.3.1). Thus, both the correlations of SDs of pitch and duration are smaller. Note that for this test we assumed normally distributed values, which is a simplification to the actual distribution.

song and the generated song are compared by the respective evaluation measures. In order to retrieve representative results, this is done 10 times for all of the 20 songs. The presented evaluation results are either presented separately for each song or as mean values of those 10 times 20 runs.

6.4.1.1 Standard Deviation

The standard deviation of pitch and duration in training data and generated data is compared to gain insight into the ability of the model to generate data which is statistically similar to the training data. Note that this only an approximate evaluation, because we assume normally distributed note events, which can usually not be found in music. Table 6.2 shows the correlation coefficients and the p-values for the respective models. Refer to the table's caption for more information.

6.4.1.2 Cross-Entropy

Cross-entropy is used in [Pearce and Wiggins, 2004] to evaluate an n-gram prediction model for musical sequences. It is a measure of the uncertainty of a prediction model, where higher values mean higher uncertainty. Let $\mathbf{e}_{t-j+1}^{t}$ be a sequence of length j, then the cross-entropy using a model m is defined as

$$H_m(p_m, \mathbf{e}_1^j) = -\frac{1}{j}\sum_{i=1}^{j} log_2 p_m(e_i \mid \mathbf{e}_1^{i-1}), \tag{6.4}$$

Setup	Pitch	Onset Pitch	Chord	Onset Chord
ORIG	**0.46052**	**0.27803**	0.20592	0.07729
SMALLR$_6$	0.49116	0.39370	0.26007	**0.04284**
SMALLR$_5$	0.48010	0.33278	**0.19711**	0.12195

Table 6.3: Mean values of cross-entropies assigned by the trained model to the training data, where lower values mean a higher prediction certainty. The cross-entropies of the setup SMALLR$_6$ show that reducing the size of region R_6 reduces the overall certainty of the model. In addition, the certainty of both the pitch sequences and the pitch onset sequences deteriorates. This might be the case because there is a connection from mirror M_2 to region R_5 (see Figure 6.4). Therefore, reducing the size of region R_6 yields higher uncertainty in both the prediction of pitches and the prediction of pitch onsets. The setup SMALLR$_5$ yields worse results mainly for pitch onset sequences, while the predictions for pitches are not strongly influenced.

where $p_m(e_i \mid \mathbf{e}_1^{i-1})$ is the probability the model assigns to event e_i given its preceding context. Usually, the cross-entropy is calculated based on a hold-out set after the model was trained on a big corpus of data. As in my implementation, the model creates almost random output if trained on the whole corpus, the cross-entropy is calculated based solely on the original song after the model was trained on that song. Table 6.3 shows results of that evaluation.

6.4.1.3 Identical Subsequences

In this section, the capability of the model to build higher-order chains of events is evaluated. For that, we use histograms to display the co-occurrence counts of identical subsequences with different lengths in melodies of the original music and the sampled counterparts. Again, we use the three different setups as described in Section 6.4.1. Figure 6.5 shows the results of the original model (ORIG). Figure 6.6 and Figure 6.7 show the histograms of the model with a reduced size of region R_6 (SMALLR$_6$) and R_5 (SMALLR$_5$), respectively. The results are mean values over 20 songs. For a discussion, refer to the captions of the respective figures.

6.4.2 Qualitative Evaluation

In music, different events like pitch, chords and their durations are temporally and spatially[2] correlated. It was the goal of my experiments to find such correlations in the generated data, too. Results of the cross-entropy and histogram evaluation presented in Section 6.4.1 show that the model is able to produce temporally correlated outcomes. However, it is difficult to judge music based solely on statistical

[2]A spatial correlation is the correlation of events which are occurring simultaneously.

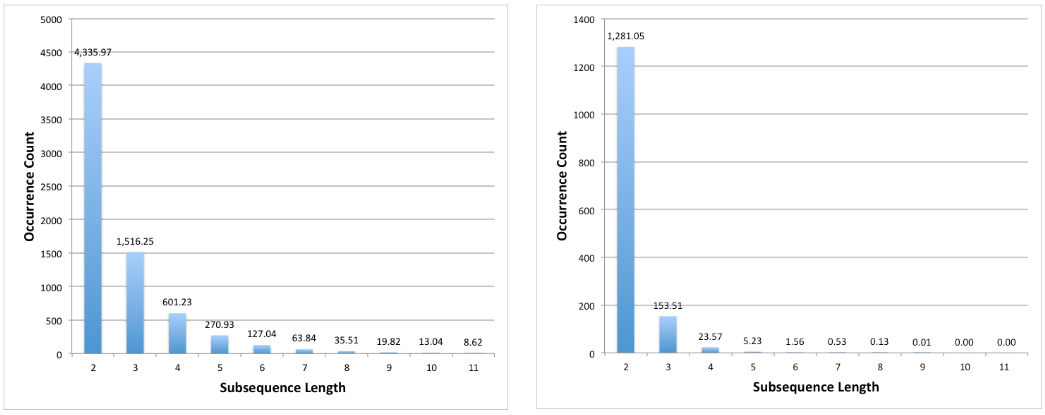


Figure 6.5: ORIG: Histograms of co-occurrences of subsequences with different lengths between original musical pieces and generated data. **Left:** Histogram of co-occurrences of pitch subsequences. **Right:** Histogram of co-occurrences of pitch onset subsequences. The model's ability to reproduce exact subsequences decreases exponentially with the length of the subsequences. As melodic lines are more distinctive than sequences of onset times in the training data, melodies were more sparsely stored than their rhythms. Therefore, identical subsequences of notes (left) tended to become longer than such of rhythms (right).

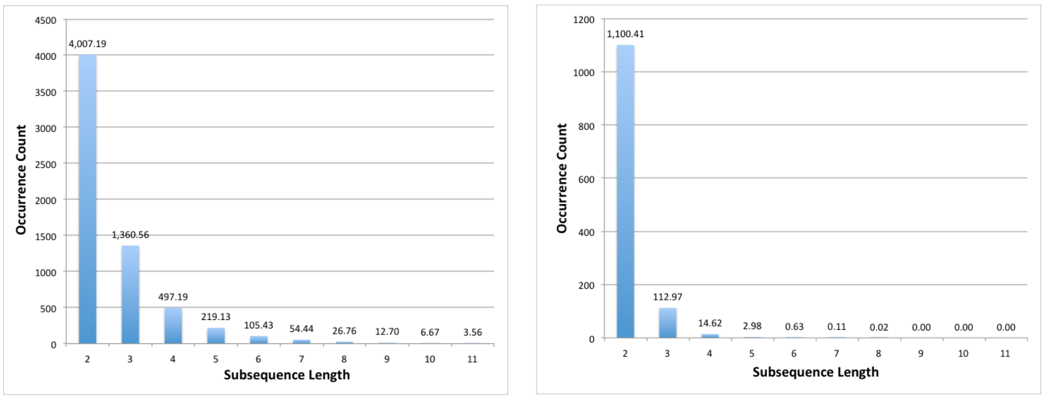


Figure 6.6: SMALLR$_6$: Left: Histogram of co-occurrences of pitch subsequences. **Right:** Histogram of co-occurrences of pitch onset subsequences. In comparison with the setup ORIG (see Figure 6.5), the ability to produce subsequences identical to subsequences in the training data is reduced. Interestingly, reducing the size of region R_6 has influence on both the pitch and the pitch onset sequences. This might be caused by the connection of M_2 to R_5 (see Figure 6.4), which constrains pitch onset sequences with pitch sequences.

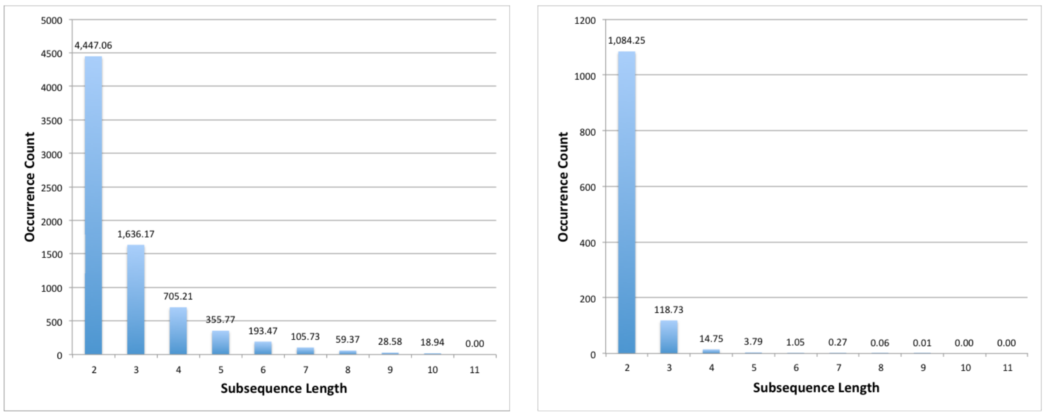


Figure 6.7: SMALLR$_5$: Left: Histogram of co-occurrences of pitch subsequences. **Right:** Histogram of co-occurrences of pitch onset subsequences. The reduction of region R_5 results in fewer co-occurrences of pitch onset sub-sequences, while the ability to create high-order chains of pitches is preserved as there is no input from pitch onset regions to a pitch region.

measures. Therefore, in this section three examples on how the model is basically able to produce meaningful outcomes are shown. We discuss three examples of generated music, which distinctively show some important correlations between training data and sampled output. The musical events are displayed in a piano roll view. Figure 6.8 shows three sections of the song ”Dock of the bay”, Figure 6.9 shows the whole song ”A hard days night”, in original and in the sampled version. Figure 6.10 shows a short section of a generated example of ”No women no cry”. The respective discussion can be found in the corresponding caption of each figure.

6.4.3 Discussion

The HTM was able to learn and generate monophonic melodies and chords to some degree. However, those results are informative to only a limited extent. First, the HTM was trained on very little data, as it tended to create almost random sequences when trained on the whole corpus. Therefore, I decided to train the model on only one song at a time and subsequently sample from it. This, however, led to a very sparse model where it was not possible to test its generalization ability. Second, generalization is also difficult to achieve, because there are only binary weights assigned to the model’s connections. Therefore, to reconstruct a state of the model in a lossless manner, all combinations of inputs have to be stored distinctively, which in turn leads to overfitting of the SP. Third, I could not make the TP work as described by Numenta. Variable-order memory could only be achieved by using a rough workaround to the online training of the TP (see 6.2.1). Since with

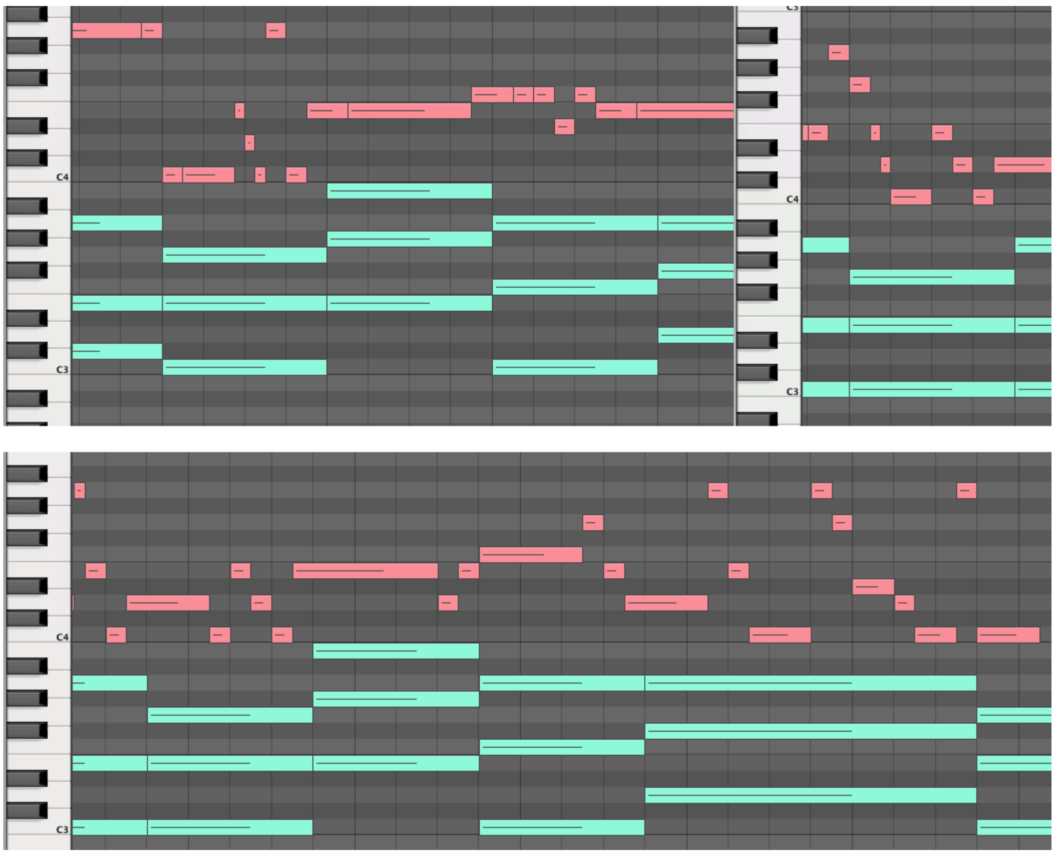


Figure 6.8: The figure shows three sections of the song "Dock of the bay". The upper part of the figure shows two sections of the original song, and the lower part is a generated example. We can see that the sampled chord sequence in this part is almost identical to that in the original song. The melody adapts to that sequence in that it has a strong tendency towards the root of the respective chord. The second, third and fourth longer notes in this example reflect exactly the root note of the corresponding chord, and also the last C Major chord is accompanied by a C note in the melody. Rhythmic properties of the original song are also preserved, as well as pitch motives. I chose to model pitch and rhythm separately, therefore we can find some pitch sequences where the rhythm differs from the original, and rhythmic passages where the pitches are substituted. As there is a connection from the pitch mirror M_2 to the rhythm region R_5 in the topology of the model (see Figure 6.4), we can also find short sequences where both pitches and their durations are identical to those of the original song.

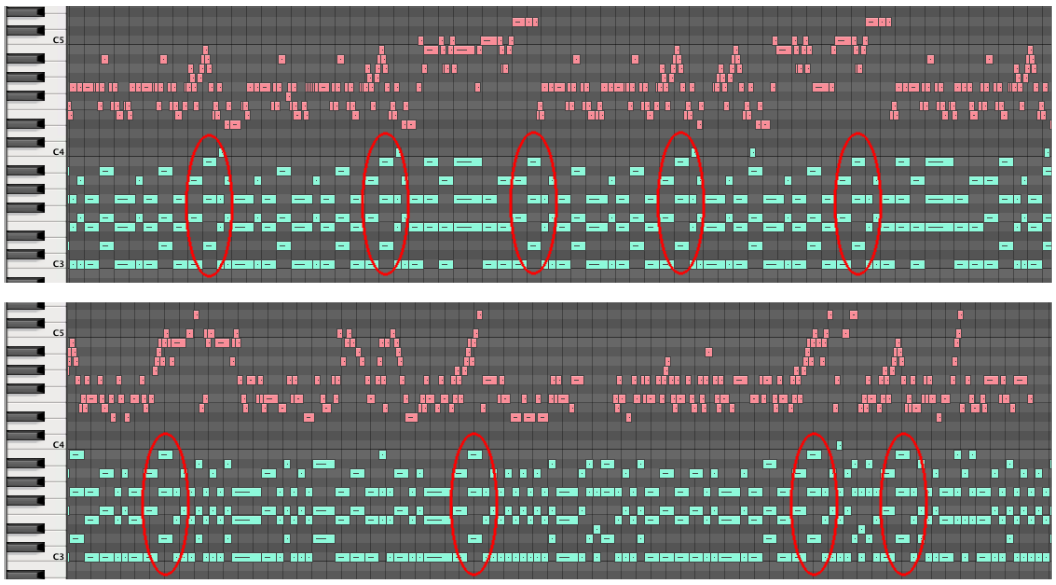


Figure 6.9: The figure shows the whole song ”A hard days night”. The upper part is again the original song, while the lower area shows the sampled counterpart. Some mid-level structures of the original song can be found in the sampled song, too. As expected, high-level structures are not reflected in the sampled example, because the memory of the model is limited. Note that the melody tends to ascend when there is a G Major chord in both the original and the sampled piece (marked with red ellipses). Such desired correlations constitute the character of a song and help the listener to recognize a piece.

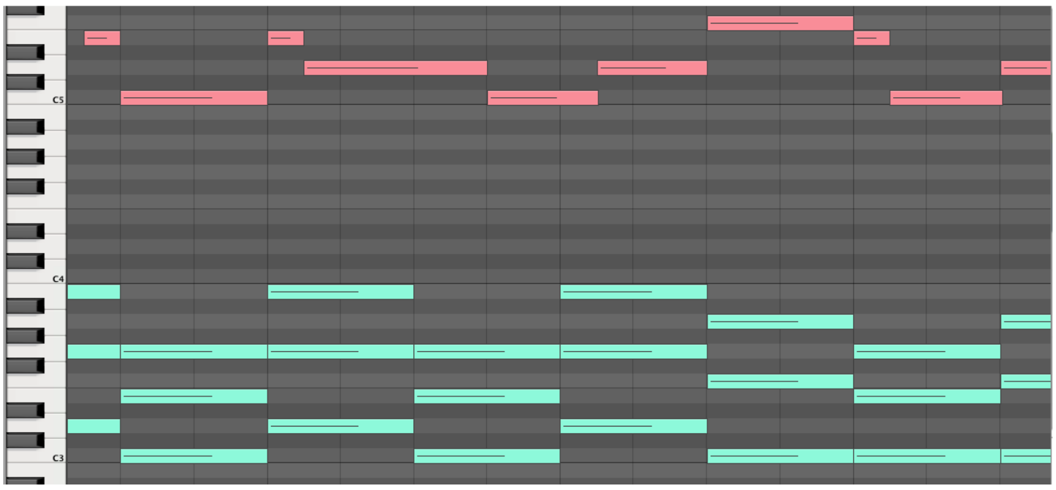


Figure 6.10: The figure shows a sampled section of the model trained on ”No woman no cry”. I chose that example because it is obvious how the melody adapts to the key note of the respective chord, which is a characteristic present in the original song, too.

that modified training the input representations had to be in a one-out-of-n form, similarities between chords which have pitches in common could not be preserved. This again impeded a possible generalization ability of the model.

Despite those limitations, the model was still able to produce some meaningful output. The histograms presented in Section 6.4.1.3 show that the model with the modified TP training was able to reproduce variable-order sequences of the original pieces. Those results, however, do not prove that the model was able to exhibit variable-order memory, as common first-order Markov chains can also reproduce longer subsequences which can be found in the training data. To what extent the tested HTM was superior in doing so is difficult to quantify formally, because sparsity in data would have to be taken into consideration.

The examples of the qualitative evaluation (see 6.4.2) show that the model is able to reproduce low-level structures, and to some extent mid-level structures of a song, and that correlations between chords and pitches, as well as onset times of notes and chords tended to get preserved.

The comparison of standard deviations in Section 6.4.1.1 shows that some of the statistical characteristics between training data and generated data match. However, for this test I assumed data which is normally distributed, which is a rough simplification.

When listening to the music the model generates after it was trained on a single song, some interesting properties are audible. First, the song the model was trained on can be recognized with considerable certainty. Due to their sparsity, chord progressions are reconstructed nicely, accompanied by characteristic melodic motifs. This shows that the model has developed the ability to store longer-order sequences. Second, melody notes adapt to the currently played chord, and chords may adapt to the currently played melody note. This shows that also static dependencies could be learned and reproduced by the model. Third, rhythmic characteristics of melodies and chord progressions are preserved, while sometimes pitch sequences are newly recombined with note onset sequences. Fourth, the model tends to generate ”medleys” of different parts of the song, because it has no notion of high-level structures. It can be noted, that the music does not sound random or arbitrary and that my subjective assessment is fairly positive.

Further experiments would have to be conducted in order to gain a thorough understanding of the strengths and weaknesses of the implemented HTM. As a summary, it can be stated that some weaknesses are obvious and result naturally from the models architecture, as stated above. Most of the strengths of the model, as pointed out in this thesis remain theory and are difficult to realize in practice. At least in my implementation, it was difficult to prove the generalization ability of the model and to make to TP work nicely without rough modifications which led to some limitations.

7 Ideas for Improvement

The formalization of the Spatial Pooler (SP) in a geometrical manner (see Section 3.3) naturally shows how the SP could be optimized for binary input data (see Section 7.1) and how it could be applied to real valued data (see Section 7.2). In addition, my work in the musical scope showed that it would be an advantage to retrieve the probability for any input bit of an observed input vector. Therefore, I developed a naive probabilistic approach to the SP, which allows for direct training of a code book based on bayesian likelihood instead of spatial closeness (see Section 7.3). Other ideas are to make units dependent on each other during training in order to reduce the reconstruction error (see 7.2.3), or to find the closest stored archetype given an arbitrary input vector (see 7.2.7).

Note that all proposed methods in this chapter are suggestions for improvements of the Numenta SDR or show different possibilities on how to implement weakly defined parts of HTMs. Note that the proposed methods were neither implemented nor tested. Therefore, this chapter should be considered as being a collection of ideas on how to improve the current SP in future research.

7.1 Optimizing the Numenta SDR Model

In this chapter, I will present an optimized version of the Numenta SDR model for binary input data, to overcome some of its drawbacks mentioned below. For that, the current SDR model will get hybridized with ideas of the Sparse Distributed Memory method [Kanerva, 1988], which was one of the first published methods similar to SDRs.

7.1.1 Problems with the Numenta SDRs

In addition to the inability to process inputs whose bits are all off, on-bits and off-bits have a different meaning to our model in both training and recognition. An off-bit during training means ”do not evaluate during recognition”. On the other hand, an on-bit during recognition cannot reduce the overlap of any unit. An input with all $x_i = 1$ would result in an overlap of 1 for all units. Therefore, this method is limited to problems where most input bits are off and gets the more imprecise the higher the ratio of on-bits to off-bits is.

Additionally, due to the different meaning of on-bits and off-bits, the risk of introducing bias gets increased. If an input pattern occurs where most bits are on, the overlap function of a lot of already trained units would yield 1, which are then under the illusion of detecting the pattern they are specialized on in that input. They then start adjusting their edges to better fit that new pattern and get ”kidnapped” by it. In that way, already acquired knowledge gets lost. Again, such bias can be introduced in other VQ approaches, too. But here the different meaning of 0 and 1 becomes a major drawback of that model. Boosting of rarely used QCs was introduced to work around that problem, even though it did not solve it (see Section 3.3.1.3).

However, the problem of having too many on-bits in the input vector is restricted to the direct input to an HTM only. Usually, the state of a region gets forwarded to its parent region in a way that guarantees that most bits are off (see Section 5.2). Therefore, the Numenta SP is still a valid solution for inter-region pattern recognition. Besides, Numenta converts real-valued input into a binary representation, where it is probably the case that vectors are generated where most bits are off, too. As it would be more convenient not to have to do so, a more direct approach is desired.

Another general problem of SDRs appears when reconstruction is performed. When an HTM gets trained, all units adjust the weights of their edges individually, only through comparison with the input vector. However, without knowledge about the overall configuration of the HTM during training, there will always be a non-quantifiable error in reconstruction because all units contribute equally to the reconstructed vector. Consider a region which is to be trained on a lot of 100-dimensional binary input vectors. Most of them are identical except one, which differs from the others in that 5 bits are flipped. As long as the region observes any of the identical vectors, its state remains the same. But when the outlier is observed, it might cause the state S of the region to change slightly to state S'. If that outlier is to be reproduced based on state S', most of the winning units which were specialized on the identical vectors will dominate those units specialized on the outlier. For the Numenta SDRs, that problem weights differently depending on how the outlier differs from the other vectors[1]. However, for example in the Sparse Distributed Representation for Continuous Input (SDRC) that domination is linear. That means, if state S' is to be reconstructed, the reconstruction is a weighted average of the dominating vectors and the outlier. If only a few winning units between state S' and state S are different, then only a little of the outliers character is preserved in the reconstruction of S'. In SDRC this problem is solved by weighting dimensions based on the reconstruction error (see Section 7.2.3). For other methods which consider input bits as independent given only one unit, like the

[1]I.e. if the bit flips resulted in 0s or 1s.

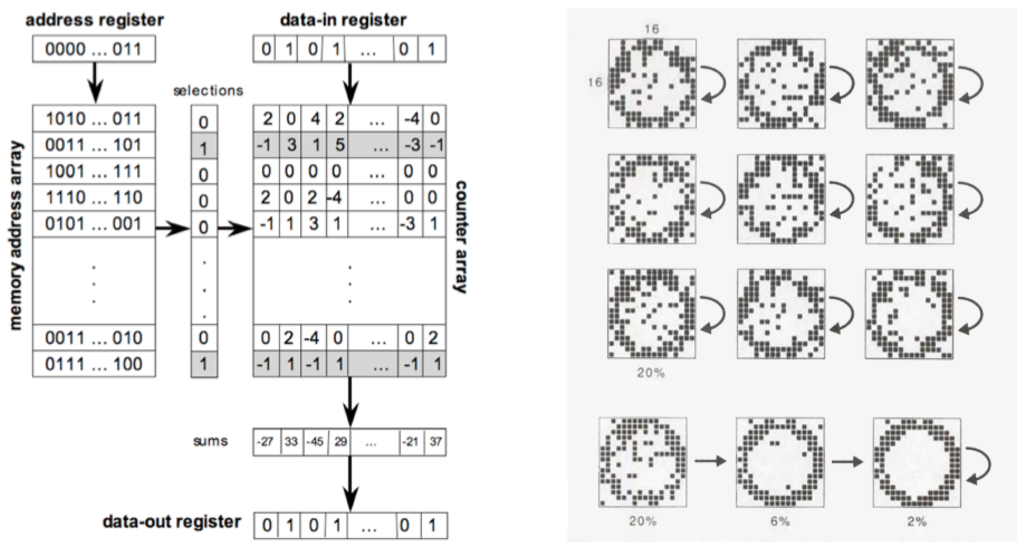


Figure 7.1: Schematic diagram of an SDM [Denning, 1989].

Naive Probabilistic Sparse Distributed Representations (NPSDR) such a solution could be also possible, even though it is not presented in this thesis.

7.1.2 Sparse Distributed Memory (SDM)

SDM, developed by [Kanerva, 1988] stores binary data patterns in a distributed manner, whereby each pattern can be addressed by a specific binary address (see Figure 7.1). At the outset, a memory address array contains rows of randomly initialized address patterns, each address pattern points to a location of a counter array. That counter array, initialized with empty counter rows, represents the actual memory where data is stored in a distributed way.

A data pattern can be stored at an arbitrary address. That address is compared to every address pattern by using the Hamming Distance. The subset of all address patterns which undercut a certain threshold determines counter row locations which are then updated with the data pattern. A counter row is updated by adding -1 to all positions which correspond to a 0 in the data pattern and by adding 1 to all positions corresponding to a 1 in the data pattern.

To retrieve a pattern given a specific address, again that address is compared to all address patterns in the memory address array and those which undercut a threshold are used for selecting counter row locations. All counters of locations selected that way are summed up vertically and a sum pattern is created. The actual data is reproduced by replacing every value of the sum pattern smaller than zero by 0 and values above zero by 1.

In this variant of SDM, the address and the data are separated from each other, even though the possibility of addressing a data pattern with itself was already investigated as the SDM was published. Figure 7.1 shows circular patterns, each of them a distorted version of a archetype and stored by inputting the data pattern in both the address register and the data-in register. By inputting yet another distorted version of the archetype into the address register and by feeding back the pattern thus retrieved into the address register, an output is generated which is almost a noise-free version of the archetype itself. In SDMs, each data pattern is stored in a sparse way in that it is distributed over several counter rows. However, in contrast to SDRs, every row considers all dimensions in both training and recognition. In the next section, I will show how we can overcome drawbacks of SDRs by utilizing some properties of SDMs.

7.1.3 A more general SDR Model

If we stored certain data patterns in an SDM (see Section 7.1.2) and if we addressed each of such patterns with itself, counter rows would tend to depict their corresponding address patterns in that a 1 in the address pattern results in a positive array position and a 0 in that pattern results in a negative array position of the respective counter row. Therefore, data rows could be initialized randomly and rows to be updated could be chosen directly if a pattern is to be stored. Hence, a memory address array would be redundant. Indeed, SDRs in HTMs operate in a very similar way. QCs are counterparts to locations in the SDM and permanences are equivalent to counters in counter rows of the SDM. In contrast to SDMs, however, each QC ignores some dimensions of the input space, which leads to less memory consumption and to higher efficiency in representation. This sparsity with respect to dimensionality is both determined by initialization and by training.

Training dimensional sparsity as described in 3.3.1.2, however, is not optimal because it leads to a different meaning of on-bits and off-bits in the input vector (see Section 7.1.1). This raises the question of how this problem could be solved in a more general way. If permanences of QCs behaved like counters in SDM, on-bits and off-bits would again have the same meaning of being just two directions in space. Permanence values would then be updated like

$$p_{ij}^{t+1} = \begin{cases} p_{ij}^t + 1 & \text{if } x_i w_{ij} = 1, \\ p_{ij}^t - 1 & \text{if } x_i w_{ij} = 0, \end{cases} \tag{7.1}$$

and the overlap function would consequently change to

$$O(\mathbf{x}, \mathbf{c}_j) = \frac{\sum_{i=0}^{N} \mathbf{1}(\frac{\text{sign}(p_{ij})}{2} + 1 = x_i) w_{ij}}{\sum_i w_{ij}}, \tag{7.2}$$

where $\mathbf{1}(\cdot)$ is the indicator function. That overlap function can be seen as the inverted and normalized Hamming Distance, in particular $N -$ Hamming Distance, for all connected edges $w_{ij} = 1$ divided through the number of connected edges. Sparsity in dimensions can be partly ensured at initialization by connecting edges of QCs to input positions in a sparse way. For an additional cutting of edges during training, I propose the following:

$$w_{ij} = \begin{cases} 1 & \text{if } |p_{ij}| > \phi_j, \\ 0 & \text{if } |p_{ij}| \leq \phi_j, \end{cases} \tag{7.3}$$

$$\phi_j = \frac{v \sum_i p_{ij} w_{ij}}{\sum_i w_{ij}}, \tag{7.4}$$

where v is the "undeterminedness factor", which defines if a dimension of a trained pattern is not informative enough to be taken into consideration. In other words, an edge is regarded to be disconnected if the magnitude of its permanence value is considerably smaller than that of other edges of its respective QC. Instead of flagging unimportant dimensions explicitly by inputting off-bits during training, patterns can then be trained mixed with other patterns or with some noise in unimportant dimensions to hold the absolute value of their permanences low. The higher the challenge during training, the higher is the robustness at recognition.

To avoid having to decide for a threshold ϕ at all, the weight of the overlap function 7.2 can also be calculated as

$$w_{ij} = \frac{|p_{ij}|}{\max \operatorname{abs}(\mathbf{p}_j)}, \tag{7.5}$$

where $\max \operatorname{abs}(\mathbf{p}_j)$ is the maximal absolute value of vector $\mathbf{p}_j$, which is the vector of all permanences of QC $\mathbf{c}_j$.

The value x_i of a dimension i of an input to reconstruct is calculated as

$$x_i = \begin{cases} 1 & \text{if } \sum_j p_{ij} w_{ij} \geq 0, \\ 0 & \text{if } \sum_j p_{ij} w_{ij} < 0. \end{cases} \tag{7.6}$$

Having set up a model that way, the advantages of both SDMs and SDRs are combined. Note that none of the described methods above were tested yet. The purpose of those suggestions is to provide ideas for future research on HTMs.

7.1.4 The Algorithm

We design the algorithm in a way that at every iteration training and reconstruction are performed simultaneously. Hereby, the current progress of training can be monitored by comparing the inputs to the reconstructed outputs. If desired, certain input dimensions can be declared as query dimensions, whose values are unknown and should therefore be reconstructed based on the internal state S of the region.

Assuming an already trained region, we start at step 4 and step 6 is to be skipped.

---

1. Sparsely connect QCs $\mathbf{c}_j$ of a region R to input dimensions $i = \{1, \dots, N\}$ of the input space $\{0, 1\}^N$, by permanently setting most weights $w_{uj} = 0$.

2. Randomly initialise all $p_{ij} \in \mathbb{N}, -h < p_{ij} < +h$. In this model, h can be interpreted as the learning rate, where a higher value of h leads to a slower convergence.

3. Calculate weights $w_{ij}, i \neq u, i \neq q$, following Equation 7.3 or Equation 7.5.

4. Temporarily set weights $w_{qj} = 0$ of all edges connected to query dimensions q of all QCs $\mathbf{c}_j$ of region R.

5. Compute state $S^t = \{c^n(\mathbf{x}^t)\}$ of region R following Equation 7.2.

6. Train region R following Equation 7.1.

7. Reset weights w_{qj} to their values before step 4.

8. Reconstruct all input values x_i following Equation 7.6.

9. Set $t = t + 1$ and go to step 4.

7.2 Sparse Distributed Representations for Continuous Input

In this section, I will describe possible approaches to properly represent continuous input patterns like grayscale images, audio spectra and other kinds of discrete functions. Indeed, many ML problems are based on processing real-valued features, but the general goal of most ML methods is to estimate one label based on a number of features. For example, given the picture of a cat, it is possible to retrieve the label ”cat”, but given the label ”cat”, it is not possible to retrieve a picture of a cat. Deep learning methods like RBMs [Hinton *et al.*, 2006] are able to do so, and so are HTMs.

What we have learned from previous sections is that SDRs can be viewed and formalized geometrically. Permanence values p_{ij} define the position of QC $\mathbf{c}_j$ in the i-th dimension of the input $\mathbf{x}$. Their weights w_{ij} define if a connection from $\mathbf{c}_j$ to

x_i exists at all[2] or how discriminative dimension i is[3]. From here it is just a small step to generalize those methods for continuous input. We will call such a memory Sparse Distributed Representations for Continuous Input (SDRCs).

7.2.1 Distance Measures in SDRCs

A common distance measure for two points in space is the Euclidean Distance (ED). The state of a region $S = \{c^n(\mathbf{x})\}$ (see Section 3.2) would then be the set of the n QCs which minimize the function

$$D(\mathbf{x}, \mathbf{c}_j) = \frac{\sqrt{\sum_i w_{ij}(x_i - p_{ij})^2}}{\sqrt{\sum_i w_{ij}}}. \tag{7.7}$$

Depending on the input data, the Manhattan Distance (MD) could be a better choice. This would result in a distance function as

$$D(\mathbf{x}, \mathbf{c}_j) = \frac{\sum_i w_{ij}|x_i - p_{ij}|}{\sum_i w_{ij}}. \tag{7.8}$$

To decrease the error for further processing state S, additionally to the position of each QC $\mathbf{c}_j$, stored in its $\mathbf{p}_j$ vector, the inverted normalized distance can be assigned as an activation value a_j of $\mathbf{c}_j$. The closest $\mathbf{c}_j$, $c^1(\mathbf{x})$ receives activation 1, and that activation decreases linearly for each $\mathbf{c}_j$, as it is further away from $\mathbf{x}$. Note that for every new observation, the activation value of each winning QC changes. In training and recognition instead of using binary values for winning and non-winning QCs, the activation value can be passed on to higher regions[4] which helps to increase the overall precision of representations of patterns (see Section 5.2).

7.2.2 Training

Training the model based on the online Lloyds Algorithm (see Section 3.2.3.1) would result in redundant QC positions if all $c^n(\mathbf{x}), n > 1$ were trained at once, because at the first iteration, the n QCs closest to $\mathbf{x}$ would be placed exactly at the position of $\mathbf{x}$ and from there they would not get separated any more. However, the region's state S can be retrieved by $S = c^n(\mathbf{x})$, and only the closest QC gets trained. By this means, that algorithm would again be appropriate for SDRCs.

Another method is the common VQ training algorithm (see Equation 3.3). However, that method is also meant to be used for $n = 1$ and therefore it is also not

[2]if $w_{ij} \in \{1, 0\}$

[3]if $w_{ij} \in [1, 0] \subset \mathbb{R}$, see 7.5

[4]In this case, also higher regions have to process the input with the SDRC method.

possible to train multiple QCs at once to minimize the MQE. We could still train only the closest QC of a region's state, as mentioned above.

Training the model with the Numenta SR method (see Equation 3.7) is another possible approach, and it works fine for $n > 1$ concerning minimizing the MQE. Depending on the step width δ, the training might take longer than in the two methods above. I suppose to decrease δ during training, as this results in faster training in the beginning and higher accuracy towards the end of the training.

To make the VQ training algorithm applicable to simultaneous training of multiple QCs, I propose to sort the closest n QCs by their distance to the newly observed data point and to divide the learning rate λ by the respective rank. For that, Equation 3.3 gets modified as

$$\mathbf{c}_j^{t+1} = \mathbf{c}_j^t + \frac{\lambda}{r_j}(\mathbf{x} - \mathbf{c}_j^t), \tag{7.9}$$

where r_j is the rank of QC $\mathbf{c}_j^t$. For the QC $\mathbf{c}_k^t$ closest to $\mathbf{x}$, rank $r_k = 1$, and the farthest QC will receive rank n.

Another proposal for training multiple QCs at once is to weight the step width of each QC with its average distance to its assigned data point. Thus, if a QC is already close to a cluster, it will not move much any more in future updates:

$$\mathbf{c}_j^{t+1} = \mathbf{c}_j^t + \frac{d_j^{\text{avg}}}{d^{\text{max}}}(\mathbf{x} - \mathbf{c}_j^t), \tag{7.10}$$

where d_j^{avg} is the average value of distances $(\mathbf{x} - \mathbf{c}_j^t)$, updated every time when $\mathbf{c}_j \in S$ and d^{max} is the highest expected distance a QC can have to any data point.

7.2.3 Weighting Dimensions

Dimension weighting is an important factor for SDRCs, because it increases robustness in recognition. It enables an SDRC to learn archetypes, even if they occur mixed in one input vector.

As an example consider the task of training an SDRC region to recognize typical patterns of piano and violin in a spectrogram. This can be done in a supervised way by providing the region with an input, which is a concatenation of a label vector and the spectrogram. Without dimension weighting it would be necessary to train the region on either solo piano or solo violin, but it would not be possible to train it on mixed examples. The goal in particular is that in the recognition process, we want the region to detect the respective instrument in a mixed signal without being disturbed by other instruments playing simultaneously.

If the region got trained on solo instruments without cutting connections to some dimensions, or weighting dimensions continuously, the region would get confused by dimensions which had low spectral energy at training and which are full of noise at recognition. However, if the region were trained on solo instruments, where dimensions with low energy would have the meaning of being less important[5], connections to such dimensions would be cut:

$$w_{ij} = \begin{cases} 1 & \text{if } p_{ij} > \gamma, \\ 0 & \text{if } p_{ij} \leq \gamma, \end{cases} \tag{7.11}$$

where γ is the "unimportance" threshold. In this case, a region would not get confused at recognition, but to explicitly instruct the region to cut some connections, as pointed out, training on solo instruments would be necessary.

With dimension weighting a unit should learn to ignore dimensions which are not informative for the feature it specializes on during training. This should lead to an even better specialization, less disturbance in recognition, and an improved precision in reconstruction.

Without dimension weighting, if the state of a region is computed, the closeness of any unit to the current observation is calculated. In reconstruction, the positions of those states are averaged. The fact that all of the *active* QCs participate with equal weights in that average calculation introduces an error. There might be a feature which is very dominant but seldom and therefore only covered by one of a set of winning units. In reconstruction, that unit is dominated by the quantity of the other units, but they do not cover that feature. The goal is to increase the weights of those dimensions of the special unit which are most informative for its special feature. This should lead to a better reconstruction and recognition, especially when there are non-linear relationships in the input space.

Without dimension weighting, a winning unit is trained without considering the state of any other winning unit. A unit is simply moved closer to dense areas in the input space. In dimension weighting, we do consider both the position and weights of the other units. That way, we make units depend on each other which should result in a better coverage of the whole input space, because units are not only pulled towards dense areas, they also start to ignore such dimensions where they are causing undesired disturbance.

An RBM can get trained through contrastive divergence [Carreira-Perpinan and Hinton, 2005]. We borrow that technique for our considerations on how to compute dimension weights. In the first step, based on an observation $\mathbf{x}$ the state S^1 of a region is computed. Then all weights of the edges of all winning units $\mathbf{c}_a \in S^1$ are updated following the equation

$$\Delta w_{ij} = \lambda e^{\log(\frac{1}{|p_{ij} - x_i|})}, \tag{7.12}$$

[5]That is similar to on-bits and off-bits have different meaning in the binary case, see 3.3.1

where λ is the learning rate. This equation increases the weights of a unit's edges to dimensions in proportion to the closeness of the unit to the respective dimension.

Having positively updated our weights we need to negatively update weights which are not helpful in describing our data. For that we reconstruct an input based on state S^1. Using this reconstruction as an observation, we calculate another state S^2. All weights of winning units $\mathbf{c}_a \in S^2$ are then updated as follows

$$\Delta w_{ij} = -\lambda e^{\log(\frac{1}{|p_{ij} - x_i|})}. \tag{7.13}$$

By comparing the observation with the reconstruction we can unveil the error introduced by training all units independently. Reducing that error by adjusting dimension weights introduces dependencies between units and ensures diversity in specialization over all units.

7.2.4 Input Normalisation

For some problems, the ratio between input dimensions is of higher importance than the absolute values. Images, for example, might be of different brightness or contrast, even though they show the same object. If, before the distance calculation, each QC normalizes the subspace it is responsible for, such differences are equalized. It is stated often in Numenta publications that images are divided into several inputs for several regions before they get combined further on in higher hierarchical layers. In such topologies, an HTM were robust with respect to differences in lighting within an image.

Each QC $\mathbf{c}_j$ normalizes the values $x_m \in \mathbf{x}$ of all connected input dimensions m, where $w_{mj} > 0$ before distance calculation as

$$\text{norm}(x_m, \mathbf{c}_j) = \frac{x_m}{\max_{x_m}(\mathbf{x})}, \tag{7.14}$$

where $\max_{x_m}(\mathbf{x})$ is the highest x_m in input vector $\mathbf{x}$.

7.2.5 Reconstruction

In order to reconstruct data based on the region's state S, for regions trained with the SDRC method, the average of values for each dimension is to be calculated. In the best case, a data point is surrounded by QCs and by averaging their positions, the position of the data point can get reconstructed again. The value x_i of a dimension i of an input to be reconstructed is calculated as

$$x_i = \frac{\sum_k p_{ik} w_{ik}}{\sum w_{ik}}, \tag{7.15}$$

for all k, $\mathbf{c}_k \in S$.

If in state S, the activation of each $\mathbf{c}_k$ is available, too (see Section 7.2.1), the weighted average can be retrieved following the equation

$$x_i = \frac{\sum_k p_{ik} w_{ik} a_k}{\sum w_{ik} a_k}. \tag{7.16}$$

7.2.6 The Algorithm

We design the algorithm in a way that at every iteration training and reconstruction is carried out simultaneously. In this way, the current progress of training can be monitored by comparing the inputs to the reconstructed outputs. If desired, certain input dimensions can be declared as query dimensions, whose values are unknown and should therefore be reconstructed based on the internal state of the region. Assuming an already trained region, we start at step 5 and step 7 is to be skipped.

---

1. Sparsely connect QCs $\mathbf{c}_j$ of a region R to input dimensions $i = \{1, \ldots, N\}$ of the input space $\mathbb{R}^N$, by permanently setting most weights $w_{uj} = 0$.
2. Randomly initialise all p_{ij}.
3. If needed[6], initialize all $d_{ij}^{\text{avg}} = 1$ and all $d_j^{\text{avg}} = d^{\max} = \epsilon$, where ϵ is the highest expected distance from any QC to any input vector.
4. Calculate weights $w_{ij}, i \neq u, i \neq q$, following 7.2.3.
5. Temporarily set weights $w_{qj} = 0$ of all edges connected to query dimensions q of all QCs $\mathbf{c}_j$ of region R.
6. Compute state $S^t = \{c^n(\mathbf{x}^t)\}$ of region R following 7.2.1.
7. Train region R following 7.2.2, if needed, update d_{ij}^{avg} and d_j^{avg}.
8. Reset weights w_{qj} to their values before step 5.
9. Reconstruct all input dimensions x_i following 7.2.5.
10. Set $t = t + 1$ and go to step 4.

---

[6] If methods are chosen that use those variables

7.2.7 Sampling from HTMs

Given an input vector $\mathbf{x}$ as a "seed" input, it can be desired to find a "better solution", which can be thought of as moving towards a close cluster in the feature space. This could be used to clean noisy data, reconstruct incomplete data, or to retrieve an unambiguous prediction of a region. A region might predict more than one state at a time. Those states are distributed over the whole memory and there is no direct way to separate multiple predicted states. Using such a mixture of predictions as a "seed"-input and sample a single unambiguous solution is one way of dealing with that problem. Throughout sampling, ensure to allow only *predicted* units to win.

This method uses a back and forth procedure, similar to finding a thermal equilibrium in an RBM and is applicable for both binary and continuous SDRs. It is also possible to clamp some bits in the input vector, which makes it possible to find unknown values while leaving known values unchanged. First tests have shown that this algorithm works well in finding archetypical solutions given a random initial input.

1. Given input $\mathbf{x}$, compute the set of winning units $\mathbf{c}_a$.
2. From $\mathbf{c}_a$, remove the worst winning unit. That is the unit in $\mathbf{c}_a$, which maximizes the distance function.
3. Reconstruct $\mathbf{x}'$, based on the residual set $\mathbf{c}_a$.
4. If $\mathbf{x}' = \mathbf{x}$, quit the loop.
5. If there are clamped positions, update those positions with their initial values.
6. Let $\mathbf{x}'$ be the new input $\mathbf{x}$ and go to 1.

7.3 Naive Probabilistic Sparse Distributed Representations (NPSDR)

In all models described so far, geometrical distance measures were used. In order to naturally work on probabilities, I developed NPSDR, a statistical approach which, in theory, has a few convenient properties. Note that the proposed method was neither implemented nor tested. Therefore, it should be considered as being a suggestion on how to improve the current SP in future research.

In this model, probabilities are calculated by counting occurrences and co-occurrences of units and input bits. Thus, information about any input pattern gets preserved, even if it occurred very rarely. In geometrically driven methods like the original SP, information is compressed into archetypes and bits which do not belong to the typical pattern are regarded as being noise. However, problems arise in domains where noise does not occur. For example, if the input consists of musical events, any of them have a meaning but some might occur very rarely and should not be considered as being noise. In NPSDR, an archetype is a probability density function (PDF). Thus, one might say that data is stored in a ”softer” way than in geometrical models.

Another convenient property is the possibility of providing PDFs as a direct input to the HTM. Many problems with real-valued data can be expressed that way and are therefore applicable to NPSDRs.

Additionally, in NPSDR boosting is not necessary. The way $c^n(\mathbf{x})$ is computed, patterns are stored in a naturally well distributed way over all units (see Section 7.3.2).

7.3.1 The Unit's Likelihood

We assume conditional independence between values x_i of an input vector $\mathbf{x}$ given that a unit $\mathbf{c}_j$ occurs. Since units are not fully connected to the input layer, this independence assumption is a rough simplification. Therefore, we use the additional descriptor ”Naive”. For NPSDR, we say that a unit occurs if it is a winning unit and thus *active*. Based on an observation $\mathbf{x} = \{x_1, \ldots, x_N\}$, the likelihood of a unit $\mathbf{c}_j$ to occur is calculated as

$$L(\mathbf{c}_j \mid \mathbf{x}) = \sqrt[s_w(j)]{\prod_i p(x_i \mid \mathbf{c}_j) w_{ij}}, \tag{7.17}$$

for all i where $w_{ij} > 0$, and

$$p(x_i \mid \mathbf{c}_j) = \begin{cases} \frac{o_{ij}}{o_{\mathbf{c}_j}} & \text{if } x_i, \\ 1 - \frac{o_{ij}}{o_{\mathbf{c}_j}} & \text{if } \neg x_i, \end{cases} \tag{7.18}$$

$$s_w(j) = \sum_i w_{ij}, \tag{7.19}$$

where $w_{ij} = \{1, 0\}$ determines if a connection between unit $\mathbf{c}_j$ and input bit x_i exists, $o_{\mathbf{c}_j}$ is the count of unit $\mathbf{c}_j$ being *active*, and o_{ij} is the co-occurrence count of unit $\mathbf{c}_j$ and input bit x_i. Taking the $s_w(j)^{\text{th}}$ root of the probability's product results in a normalization, which is only necessary, if the amount of connected edges over all QCs varies.

With this method it is possible to input the probability of x_i, if its actual state is not known. For that, we change Equation 7.18 to

$$L(\mathbf{c}_j \mid p(x_i = e)) = 1 - \frac{2eo_{ij} - o_{ij}}{o_{\mathbf{c}_j}} - e. \tag{7.20}$$

Here, we interpolate between $p(x_i \mid \mathbf{c}_j)$ and $p(\neg x_i \mid \mathbf{c}_j)$, which makes it possible to store real-valued input vectors.

If the model should be able to handle mixed patterns[7], or if most bits in the input data are off-bits, it might be better that only on-bits are taken into consideration. For that, in Equation 7.17 we consider only those i, where $x_i = 1$ and $w_{ij} > 0$. In addition, we change Equation 7.19 to

$$s_{wx}(j) = \sum_i w_{ij} x_i. \tag{7.21}$$

7.3.2 Training

Training is simply done by counting occurrences and co-occurrences of units and input bits. As in VQ methods, the model is initialized randomly and for each training step, the region's state is computed by assuming an already trained model which is then to be adjusted to fit the input data even better.

In the first step, based on the input vector $\mathbf{x}$, the state $S = c^n(\mathbf{x}) = \{\mathbf{c}_a\}$ is determined by searching for the n units which maximize Equation 7.17. We then consider all winning units $\mathbf{c}_a = \text{true}$ and all other units $\mathbf{c}_k = \text{false}, a \neq k$ and proceed with the next step. The general formulas for updating occurrence counts and co-occurrence counts in that step are as follows:

$$o^{t+1}_{\mathbf{c}_j} = o^t_{\mathbf{c}_j} + \text{eval}(\mathbf{c}_j, t)\lambda, \tag{7.22}$$

$$o^{t+1}_{x_i} = o^t_{x_i} + x^t_i \lambda, \tag{7.23}$$

$$o^{t+1}_{ij} = o^t_{ij} + x^t_i \text{eval}(\mathbf{c}_j, t)\lambda, \tag{7.24}$$

$$\text{eval}(e, t) = \begin{cases} 1 & \text{if } e, \\ 0 & \text{if } \neg e, \end{cases} \tag{7.25}$$

where o_{x_i} is the occurrence count of input bit x_i and λ is the learning rate.

7.3.3 Reconstruction

In order to reconstruct data based on the region's state $S = \{\mathbf{c}_a\}$, for regions trained with the NPSDR method, the probability of bit x_i of a pattern to be reconstructed is calculated as

[7]Mixed patterns are more than one pattern in one input vector combined with the bitwise OR operator.

$$p(x_i \mid S) = \frac{\sum_a \frac{o_{ia}}{o_{\mathbf{c}_a}} w_{ia}}{\sum_a w_{ia}}. \tag{7.26}$$

7.3.4 The Algorithm

We design the algorithm in a way that at every iteration training and reconstruction is processed simultaneously. The current progress of training can be monitored by comparing the inputs to the reconstructed outputs. If desired, certain input dimensions can be declared as query dimensions, whose values are unknown and should therefore be reconstructed based on the internal state S of the region.

Assuming an already trained region, we start at step 3, and step 5 can be skipped.

---

1. Sparsely connect QCs $\mathbf{c}_j$ of a region R to input dimensions $i = \{1, \ldots, N\}$ of the input space $\{0,1\}^N$, by permanently setting most weights $w_{uj} = 0$.
2. Randomly initialize all $o_{ij}, \in \mathbb{N}, 0 < o_{ij} < h$. In this model, h can be interpreted as the learning rate, where a higher value of h leads to a slower convergence. Randomly initialize all $o_{\mathbf{c}_j}, o_{x_i} \in \mathbb{N}, h < o_{\mathbf{c}_j}, o_{x_i} < 2h$. Typically, set $\lambda = 1$.
3. Temporarily set weights $w_{qj} = 0$ of all edges connected to query dimensions q of all QCs $\mathbf{c}_j$ of region R.
4. Compute state $S^t = \{c^n(\mathbf{x}^t)\}$ of region R by searching for the n units which maximize Equation 7.17.
5. Train region R following 7.3.2.
6. Reset weights w_{qj} to their values before step 3.
7. Reconstruct all input values x_i following Equation 7.26.
8. Set $t = t + 1$ and go to step 3.

7.4 Possible Improvements in Hierarchical Topologies

In hierarchical topologies a different kind to what was described in 5 of spatial training and reasoning in the SP could be performed. The idea is that by incorporating feedback in training, an HTM might perform better than using exclusively feed-forward input. Again, this method was not tested and is a thought experiment on how to improve the performance of HTMs. The idea is not limited to NPSDR, theoretically it can be used with any SP method.

7.4.1 Training

Spatial training can be done iteratively by using the feed-forward procedure only. Simultaneous training, however, can be an interesting alternative, because it enables regions to incorporate feedback information from their parents. In this case, the state of a region can be different even though it is provided with an identical input from below. This can be achieved by training an HTM based on states derived through reasoning as described in Section 7.4.2.

Imagine that a human being is listening to music, and somebody says ”listen to the melody”, ”listen to the beat” or ”listen to the chord progression”. Such an instruction will change the human's attention and he will be trained on different musical events, depending on the context which, in this case, is the instruction. Similarly, an HTM could be trained in a supervised manner by inputting labels like ”there is a piano playing”, or ”there is no piano playing”, while a spectrogram is fed into another input. Depending on the label different units would be *active* for the same input. I assume that this enables a region to better recognize some patterns in mixed inputs.

Additionally, multiple inputs could mutually inform each other, because, for example, by a combination of many different symbolic musical events, a genre could be implicitly derived in the upper most region of an HTM. By using information fed back from this region as it is done in reasoning would lead to different states for the same sequence of notes, and to desired different transition probabilities for the temporal training of an HTM.

7.4.2 Reasoning

In reasoning, the information derived from the feed-forward and feedback procedure gets combined. At first, feed-forward is executed (see Section 7.4.3), and the states of all regions are computed. Then, we perform a feedback procedure as described in Section 7.4.4, whereby the final probability of any QC is derived:

$$p(\mathbf{c}_j^k \mid H) = L(\mathbf{c}_j^k \mid \mathbf{x}^{k-1}, \mathbf{x}^{k-2}, \dots, \mathbf{x}^0) p(\mathbf{c}_j^k \mid S^{k+1}, \dots, S^G), \qquad (7.27)$$

where the first term is the feed-forward term, the second term is the feedback term and H is the HTM. We then update the state of any region to the set of its n most probable QCs.

7.4.3 Feed-Forward

In order to recursively compute the region's states in hierarchical topologies, the states of all regions are computed iteratively from the bottom to the top. One such iteration is shown in Section 7.3.1. This results in $L(\mathbf{c}_j^k \mid \mathbf{x}^{k-1}, \mathbf{x}^{k-2}, \dots, \mathbf{x}^0)$ for any QC $\mathbf{c}_j^k$, where k is the number of a hierarchy level.

7.4.4 Feedback

In the NPSDR feedback, the probability of an input bit to occur does not only depend on the state of its parent, but also on the state of its parent's parent, and so on. For reconstruction of an input bit $x_i = \mathbf{c}_i^0$, Equation 7.26 gets extended by a recursive evaluation of the probability of connected parent QCs as

$$p(\mathbf{c}_i^k \mid S^{k+1}, \ldots, S^G) = \frac{\sum_a \frac{o_{ia}}{o_{\mathbf{c}_a^{k+1}}} p(\mathbf{c}_a^{k+1} \mid S^{k+2}) w_{ia}}{\sum_a p(\mathbf{c}_a^{k+1} \mid S^{k+2}) w_{ia}}, \tag{7.28}$$

where S^k is the state of a region in the k-th hierarchy level $k = \{1, \ldots, G)$ and $p(\mathbf{c}_a^G \mid S^{G+1}) = 1$. Because that method is to be processed from the top to the bottom of the HTM, we call it feedback.

7.5 Probabilistic Sampling from an HTM Hierarchy

With a well designed SP it is theoretically possible to sample from a HTM hierarchy. In this section, I want to perform a thought experiment on how this could be achieved. We use the NPSDR method (see Section 7.3) for this experiment, because it enables us to derive probabilities and thus it allows for statistical sampling. The state of a region depends on its input and on its probability given the parent's state. The input provides high resolution information, while parent regions provide lower resolution context information.

For example, consider that an HTM should be trained on symbolic musical events and there are two direct inputs, scale and chord. There are three regions, R_1, R_2 and R_3. As a direct input, R_1 has scale, where a binary input vector of length 12 is provided, one bit for each possible note. Notes which are part of a scale will result in an on-bit while the rest of the vector values are off-bits. In region R_2 chord information is input the same way. Chord notes will have a corresponding on-bit, all other bits are off. Finally, region R_3 combines region R_1 and region R_2.

Now, sampling can be done by either hard or soft fixing some bits, or setting them to query bits in the first place. Hard fixing means that the fixed states of input bits are fed into the model at each sampling iteration, while soft fixing means that the bit's states are input at the first iteration only. For all other sampling iterations, the bit's states are sampled based on the derived probabilities after each iteration. Query bits have no state at the first sampling iteration and become soft fixed for all other iterations.

Let us assume that we want to sample a chord which occurs in the C Major scale. Therefore, we hard fix the C Major scale, while all bits of the chord input are query bits. Then, we compute the states of all regions by executing feed-forward (see

Section 5.3). All units where Equation 7.17 yields an undefined result[8], are set to query units and will be transformed into query bits for the output of the region. This will be the case for all units in region R2. Thus, based on the output of R_1 only, the state of the region R_3 is computed.

In the next step, feedback is executed (see Section 7.4.4). First, let us look at the scale branch only. The state of region R_3 is a somehow blurred representation of the C Major scale. Because it was trained on combinations of scales and chords, that representation might be a mixture of the C Major scale with other scales which tend to occur together with typical C Major chords. That is why we referred to information of parent regions as lower resolution information. Fortunately, R_1's units' feed-forward probabilities represent a high resolution representation of the C Major scale. If feed-forward and feedback probabilities are multiplied at reasoning (see Section 7.4.2), we will again derive a good C Major scale representation in R_1. Note that with this procedure, we have not gained new information. But if contradicting sensor input would have been supplied to the HTM[9], through the intersection step, the C Major scale probabilities could have been adjusted to better fit the contradicting inputs.

Now let us process feedback in the chord branch. In the first step, based on the state of region R_3, a reconstruction in R_2 is computed. That reconstruction is a representation of all chords which could occur in the C Major scale. Reasoning is not computed, because all units were query units, but still the best n units are chosen to be *active*. If that representation is then used to reconstruct the actual direct input, that input reconstruction reflects the probabilities of all chord notes in the C Major scale. Now we can sample actual notes based on those probabilities. Due to this distribution allows for many different chords, we will not get a single final solution.

Thereafter, we re-input the chord notes thus derived and the hard fixed C Major scale. After a few feed-forward and feedback iterations, the state of the model converges to a quite stable internal representation, and a single chord which fits the C Major scale will be found with the probability of that chord to occur in a C Major scale.

[8]Equation 7.17 yields an undefined result if all edges are disconnected, because all input bits x_i are query bits, i.e. all $w_{ij} = 0$.

[9]Contradicting information would be, for example, a C Major scale with a B Flat chord, which is not part of the C Major scale.

8 Conclusion

In this thesis, I have presented my experiments on the Hierarchical Temporal Memory (HTM). The two parts of the HTM, the Spatial Pooler (SP) and the Temporal Pooler (TP) were introduced, and an evaluation on the HTM's overall performance in training and in generation of monophonic melodies and chords was given. To that end, I implemented the HTM in Java and used the Temperley Rock Corpus [De Clercq and Temperley, 2011] as training data. Finally, I presented my own ideas on how to possibly improve the SP. However, those suggestions for improvement were nor implemented neither tested.

The theoretical basis of the HTM, introduced in the book "On Intelligence" [Hawkins and Blakeslee, 2004] seems to be well-conceived and promising. The idea of hierarchical modeling of temporal sequences is a powerful concept. The HTM itself, which is the computational counterpart designed by Numenta, still seems to be immature to some degree. Modeling input data in a feature space, like it is the case in Restricted Boltzmann Machines (RBMs) [Hinton *et al.*, 2006] or in dictionary based methods [Sturm *et al.*, 2009] has already shown to work well. Combining such representations with temporal models, like Recurrent Neural Networks (RNN), is a good and logical step. This, for example, has also been done in [Boulanger-Lewandowski *et al.*, 2012] where RBMs were combined with RNNs. The humble question remains, if Numenta has really designed those mechanisms in a good way, compared to the before mentioned methods.

My own experiments have shown that the HTM is difficult to implement and that it is hard to obtain the expected functionality. In particular, it was not possible to obtain a generalization ability of the model, and for me it was not possible to make the TP training work as expected. With small training data, it was possible to generate meaningful output, however, that may follow mostly from increased sparsity where the model has no need to generalize. As examples of Numenta show, their model seems to work better, and by now, an implementation of that model called NuPIC is available at their website [Numenta, 2014]. Maybe, with that code in hand, a better examination and evaluation of HTMs will be possible in future.

Bibliography

[Boulanger-Lewandowski *et al.*, 2012] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *ICML*. icml.cc / Omnipress, 2012.

[Carreira-Perpinan and Hinton, 2005] Miguel A Carreira-Perpinan and Geoffrey E Hinton. On contrastive divergence learning. In *Artificial Intelligence and Statistics*, volume 2005, page 17, 2005.

[Coates and Ng, 2011] Adam Coates and Andrew Ng. The importance of encoding versus training with sparse coding and vector quantization. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 921–928, New York, NY, USA, June 2011. ACM.

[De Clercq and Temperley, 2011] Trevor De Clercq and David Temperley. A corpus analysis of rock harmony. *Popular Music*, 30(1):47–70, 2011.

[Denning, 1989] Peter J Denning. *Sparse distributed memory*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1989.

[Fernández *et al.*, 2011] David Rozado Fernández, Francisco B Rodrıguez Ortiz, and Pablo Varona Martinez. *Analysis and Extension of Hierarchical Temporal Memory for Multivariable Time Series*. PhD thesis, Technical University of Kaiserslautern, 2011.

[George and Hawkins, 2005a] D. George and J. Hawkins. A hierarchical bayesian model of invariant pattern recognition in the visual cortex. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 3, pages 1812–1817 vol. 3, 2005.

[George and Hawkins, 2005b] D. George and J. Hawkins. Invariant pattern recognition using Bayesian inference on hierarchical sequences. *Redwood Neuroscience Institute Technical Report*, 2005.

[George and Hawkins, 2009] Dileep George and Jeff Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS Computational Biology*, 5(10), 2009.

[George, 2008] Dileep George. *How the Brain Might Work: A Hierarchical and Temporal Model for Learning and Recognition.* PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3313576.

[Gersho and Gray, 1991] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[Hawkins and Blakeslee, 2004] Jeff Hawkins and Sandra Blakeslee. *On Intelligence.* Times Books, 2004.

[Hawkins and George, 2006] Jeff Hawkins and Dileep George. Hierarchical temporal memory: Concepts, theory, and terminology. Numenta Inc. Whitepaper, 2006.

[Hawkins, 2011] Jeff Hawkins. Hierarchical temporal memory including cortical learning algorithms. Technical report, Numenta, Inc. Ver. 0.2.1, 2011.

[Hinton *et al.*, 2006] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[Kanerva, 1988] Pentti Kanerva. *Sparse distributed memory.* The MIT Press, 1988.

[Lloyd, 1982] Stuart Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[Martinetz and Schulten, 1991] T. Martinetz and K. Schulten. A "Neural-Gas" Network Learns Topologies. *Artificial Neural Networks*, I:397–402, 1991.

[Maxwell *et al.*, 2009] James B. Maxwell, Philippe Pasquier, and Arne Eigenfeldt. Hierarchical sequential memory for music: A cognitive model. In Keiji Hirata, George Tzanetakis, and Kazuyoshi Yoshii, editors, *ISMIR*, pages 429–434. International Society for Music Information Retrieval, 2009.

[Numenta, 2014] Numenta. Inc. url: http://www.numenta.org, June 2014.

[Pearce and Wiggins, 2004] Marcus Pearce and Geraint Wiggins. Improved methods for statistical modelling of monophonic music. *Journal of New Music Research*, 33(4):367–385, 2004.

[Sahlgren, 2006] Magnus Sahlgren. *The Word-Space Model: Using distributional analysis to represent syntagmatic and paradigmatic relations between words in high-dimensional vector spaces.* PhD thesis, Stockholm, 2006.

[Schey, 2008] Nathan C Schey. Song identification using the numenta platform for intelligent computing. BSc. Thesis, The Ohio State University, 2008.

[Sturm *et al.*, 2009] Bob L. Sturm, Curtis Roads, Aaron McLeran, and John J. Shynk. Analysis, visualization, and transformation of audio signals using dictionary-based methods †. *Journal of New Music Research*, 38(4):325–341, 2009.

[Thornton *et al.*, 2008] John Thornton, Jolon Faichney, Michael Blumenstein, and Trevor Hine. Character recognition using hierarchical vector quantization and temporal pooling, 2008.

[van Doremalen and Boves, 2008] Joost van Doremalen and Lou Boves. Spoken digit recognition using a hierarchical temporal memory. In *INTERSPEECH*, pages 2566–2569. ISCA, 2008.

Appendix A

Curriculum Vitae

Stefan Lattner graduated at the Higher Technical College for wood engineering in Mödling. After civilian service, he studied one year bioinformatics in the University of Applied Sciences Hagenberg. After that, he switched to the subject media technology and design at the same campus, where he received his bachelor's degree in July 2009. In the same year, he started to work for the Re-Compose GmbH as chief developer and project manager, and began his study for the master's degree for pervasive computing in the Johannes Kepler University Linz. Since October 2013, he works at the Austrian Research Institute for Artificial Intelligence in Vienna, where he is involved in the Lrn2Cre8 project, which aims to understand the relationship between learning and creativity in the musical domain.

Appendix B

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum Unterschrift