DIPLOMA THESIS

Extending Hierarchical Temporal Memory for Sequence Classification

Klaus Greff

Technische Universität Kaiserslautern
AG Wissensbasierte Systeme

Prof. Dr. Prof. h.c. Andreas Dengel

Dr. Gunnar Aastrand Grimnes

Diploma thesis at the
AG Wissensbasierte Systeme
Technische Universität Kaiserslautern
Fachbereich Informatik

Klaus Greff

Extending Hierarchical Temporal Memory for Sequence Classification

Supervisors
Prof. Dr. Prof. h.c. Andreas Dengel
Dr. Gunnar Aastrand Grimnes

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit mit dem Thema "Extending Hierarchical Temporal Memory for Sequence Classification" selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quellen, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Kaiserslautern, 10. November 2010

Contents

List of Figures

List of Tables and Algorithms

Tables

Algorithms

1. Introduction

This thesis tackles the problem of sequence learning using Hierarchical Temporal Memory as a first step towards a framework for combined temoral and spatial inference. Sequence learning is a crucial part of intelligence. Sun [2001] even considered it "*the most prevalent form of human and animal learning*". There are many applications where sequences are the pivotal elements including natural language processing, speech recognition, video analysis, planning, robotics, adaptive controls, time series prediction, finance, DNA sequencing, compression and many more. A large variety of methods deal with all the different forms of sequence learning. They include time-series analysis, regression, compression, grammars, symbolic planning, hidden Markov models, conditional random fields, recurrent neural networks et cetera.

However, in most real world applications, the temporal component is not the only dimension of the problem. Often, there is also a spatial[1] problem to solve, like object recognition in the case of video analysis or identifying phonemes for speech recognition. While classification of spatial patterns is well studied, the combination of both, spatial and temporal classification, is not. In most cases, they are performed separately, i.e. first do the spatial classification and then do sequence learning on those classes. Well known examples are hidden Markov models and conditional random fields. Only few methods exist that combine both spatial and temporal learning in a very tight way (e.g. recurrent neural networks do this).

The problem is that useful information is lost due to this separation. Temporal classification could support spatial classification at various levels of abstraction. For example, to filter background noise or to track multiple objects and predict their mutual occlusion. Sequence learning enables the powerful ability to make predictions, which could be applied to verify the interpretation and to disambiguate input. This is, up to a certain point, also true for a simple concatenation of spatial and then temporal learning, but it would be much stronger for a real *joint inference*. Therefore, it is an important issue to figure out how to tightly combine both methods.

Hierarchical Temporal Memory (HTM) possibly offers a solution for this joint inference. It is a quite new technology (2008) inspired by the human cortex to do (spatial) classification. They have shown some promising results including CAPTCHA recognition [Hall and Poplin, 2007], content-based image retrieval [Bobier and Wirth, 2008] and spoken digit recognition [van Doremalen and Boves, 2008].

What is interesting about it is that although it is designed to do spatial classification, it uses the temporal structure of the training data. The idea can roughly be summarized as "observations that are close to each other in time are likely to belong to the same cause/object". This is used to build invariant representations at different levels

[1] In this case, spatial refers to the problem space as a generic term for all dimensions but time. They do not necessarily have to refer to space.

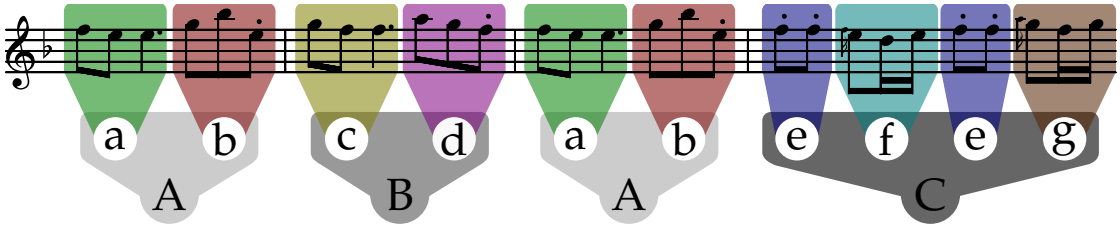


Figure 1.1.: Hierarchical structure within music. W. A. Mozart: “Eine kleine Nachtmusik”.

of abstraction within their hierarchical structure. Thereby HTMs utilize yet another advantage of the connection between spatial and temporal structure. However, when it comes to classification HTMs discard all the temporal information. So, once the training is completed, inference relies on spatial information only[2]. At this point, there is obviously room for improvement.

The goal of this thesis is to explore the possibilities of combining spatial and temporal learning by extending Hierarchical Temporal Memory. We will adopt the assumption of hierarchically structured data, which allows HTMs to have the main classification task split up into a set of smaller tasks at different levels of abstraction. This allows moving the separation of spatial and temporal learning to a much smaller scale, so their cooperation can be much closer. Another advantage of this approach is that we can potentially re-use any existing algorithms that have been developed for sequence learning and for spatial classification because we still separate those tasks.

Implicit in approach is the assumption that the data is structured hierarchically in both, space and time. Otherwise, the described way of splitting will not work. But we believe that this hierarchical structure is inherent in a wide range of real data. Temporal hierarchies can be found for example in speech which decomposes in phonemes, syllables, morphemes, words and sentences, music that can be divided into themes, periods, phrases and motifs (see also figure 1.1) and movies which consist of frames, shots, scenes and parts. Spatial hierarchies are even more obviously found: A car consists of a chassis, wheels, doors and an engine, which consists of cylinders, spark plugs, valves and so forth. A piece of music often contains different instruments and sometimes one or more voices. A typical dinner consists of an appetizer, a main course and a dessert, etc..

Considering these examples, we expect the required structure to be inherent in many interesting real world problems. Therefore a framework that provides a close cooperation between spatial and temporal learning will probably significantly improve performance on complicated problems like video analysis or speech recognition with high background noise. In general, this method could help to tackle very difficult problems that have not been solved yet.

In this thesis, we extend the theoretical framework of HTMs enabling them to do sequence classification. The improved framework is implemented and used to evaluate the algorithms on artificial data. We show this approach to be a viable first step towards a joint inference.

[2]There is in fact an attempt to utilize temporal information which is called Time-Based-Inference (TBI). But it is quite limited and not well documented.

2. Hierarchical Temporal Memory

In this chapter, we give a brief introduction to *Hierarchical Temporal Memory* (HTM) because understanding its basic concepts is crucial for the rest of the thesis. We will also discuss its problems related to sequence learning. However, we will concentrate on the basics. For a further more in-depth discussion of the algorithms, refer to Numenta's official documentation [George and Jaros, 2007] and the PhD thesis of George [2008]. The algorithms are based upon the ideas of the *Memory Prediction Framework* which is presented by Hawkins and Blakeslee [2005]. The basic concepts are explained by Hawkins and George [2006]. See Numenta [2007b] for a comparison to existing machine learning techniques like neural networks and Bayesian networks.

Note that this thesis builds upon the first generation algorithms as in the *Numenta Platform for Intelligent Computing* (NuPIC) version 1.5. This release was the last to come with a comprehensive documentation of algorithms. Changes done in the releases 1.6 and 1.7 as well as the *Fixed Density Representation* algorithms that Numenta scheduled for the 2011 are not considered here. For a detailed description of all algorithms and parameters, refer to the documentation that was included in NuPIC 1.5 release [Numenta, 2007a].

2.1. Overview

HTMs act as a classifier that can be trained in an either supervised or unsupervised manner. During training, it requires a temporally structured stream of input data while inference is done independently for every frame. A hierarchical structuring of the data in both, space and time, is crucial for the algorithm to succeed (see Numenta [2007c] for more details).

2.2. Structure

A HTM network is organized as a tree-shaped hierarchy. Although uncommon, multiple parents per node are allowed, therefore letting the receptive fields of the parents overlap (see for example node N_{14} in figure 2.1 on the next page). The input data is distributed amongst the leaves of the tree. Each node then runs some computation on this data and passes its results upwards to its parent. This process repeats until the information reaches the top of the tree. The root is also called *classifier node* because it will output the result of the inference and also, it receives the categorical information during the process of supervised learning.

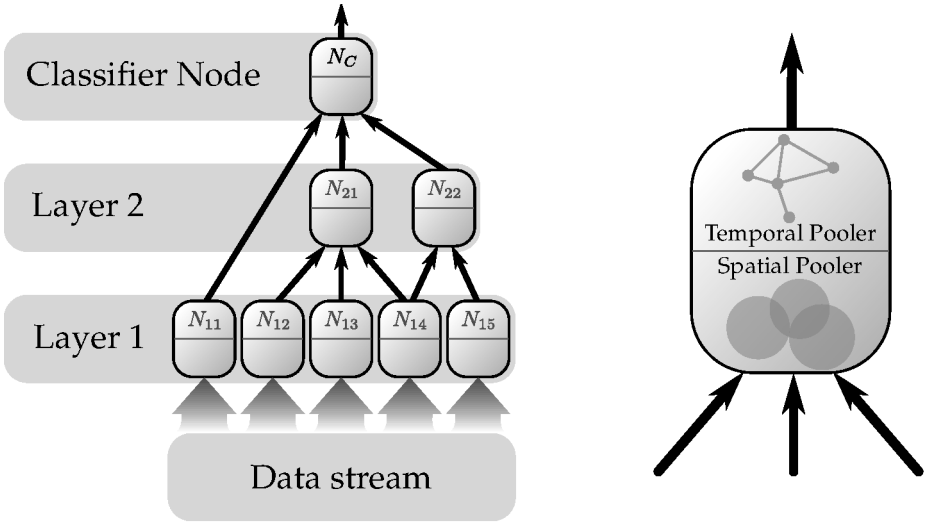


Figure 2.1.: Simple HTM network (left) and a schematic drawing of a node (right) and its internals. Note that the Network contains a generation-skip (from N_{11} to N_C) and an overlap (N_{14} has two parents).

2.3. Training and Inference

Training of the network happens layer-wise. The layers are numbered starting with the leaves as seen in figure 2.1. Once the first layer is trained, it switches into inference mode, thus producing an output for the next layer to be trained on. This is repeated until the classifier-node is reached.

In the unsupervised case the top-node is trained like every other node. If, on the other hand, categorical information is available, it can also be trained in a supervised way. Note that even in that case the classifier node is the only node to be trained in a supervised manner.

2.4. Nodes

Each node in the hierarchy is exposed to a temporal stream of data and, independent of its position[1], performs the same two tasks: finding reoccurring spatial patterns and common sequences of those patterns. These two operations are called spatial pooling and temporal pooling.

2.4.1. Spatial Pooling[2]

During training time, the node performs a clustering of the input data to reduce the possibly infinite number of different inputs to a small set of so-called *coincidences*. By

[1] Actually the spatial pooling algorithm differs for the leaves and the inner nodes.
[2] The "space" in spatial pooling refers to the problem space, which itself does not need to be spatial.

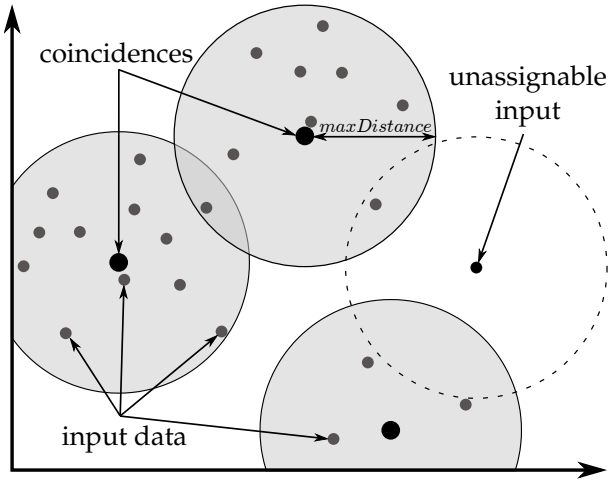


Figure 2.2.: Spatial pooling example for a 2-dimensional problem space. Three coincidences are already stored with their *maxDistance* indicated as gray circles. Also a new non-assignable input can be seen which will be made a new coincidence, given that *maxCoincidences* is higher than 3.

doing so, it becomes feasible for the next step to find reoccurring sequences in the data stream. This idea is implemented in two different ways: one for the leaf nodes and one for all the others.

The leaf nodes operate as follows: For every input that, by euclidean distance, differs more than a parameter (*maxDistance*) from every previously found coincidence, a new one is made. This is repeated until a threshold (*maxCoincidences*) is reached (compare algorithm A.1 on page 69).

Spatial pooling in the inner nodes happens in a slightly different way. Before comparing or storing, the input is sparsified in the following way: The maximum value of every output vector of each child is determined. The corresponding components are then set to 1 and all the others to 0. After that, the vectors are concatenated. Thus resulting in a vector that contains as many 1s as the node has children while all the other components are 0.

That approach is taken because the output vectors of each node (that is the output of temporal pooling) is a probability distributions. Hence, in the perfectly clean noiseless case, they would contain exactly one 1 and all the other components would be 0. Therefore, a winner-takes-it-all procedure is a reasonable technique for noise reduction (see algorithm A.3 on page 70).

2.4.2. Temporal Pooling

This step is based on the idea that patterns that frequently transit to one another are likely to belong to the same “cause”. Therefore, treating them in the same way can allow

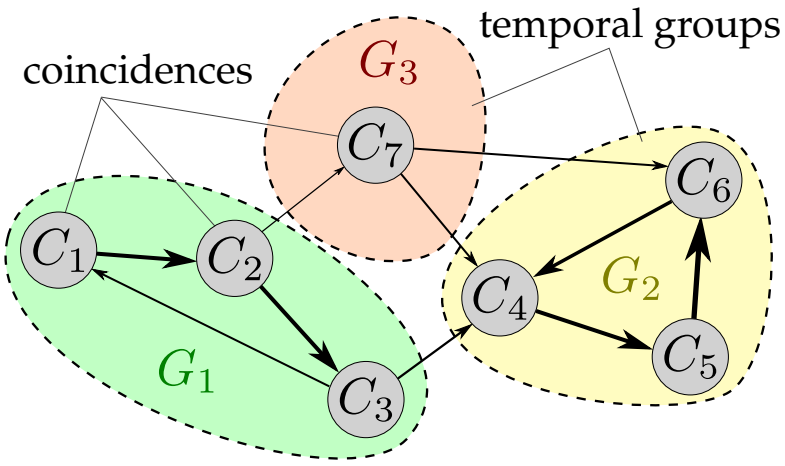


Figure 2.3.: Simple temporal pooling example with seven coincidences that are clustered into three temporal groups.

useful invariants to be formed. This is implemented by maintaining a *transition matrix* $\mathbf{T}$, with every element T_{ij} corresponding to the number of transitions from the i-th to the j-th coincidence. This matrix is then normalized forming a Markov chain that contains estimated probabilities for each transition. By using this probabilities as a measure of similarity, the coincidences are then merged using a *Hierarchical Agglomerative Clustering* algorithm. The grouping continues until a pre-defined number of groups ($maxGroups$) is reached. Those resulting sets of coincidences are called *temporal groups.*

2.4.3. Inference

During inference, every node starts by determining the distance of the new input to every coincidence. This vector of distances is, in the second step, turned into a "belief distribution" by calculating the probability that the input belongs to each coincidence utilizing a Gaussian distribution (with mean 0 and σ as fixed parameter). Note that the result is, in general, not a probability distribution because it does not necessarily sum to one.

This vector is then used to calculate a belief distribution over all temporal groups. The belief for a temporal group is determined by simply taking the maximum belief of all coincidences belonging to that group. The resulting vector is then passed upwards to the node's parent(s).

2.4.4. Supervised Training

In case of a supervised training the temporal pooling algorithm of the classifier node is substituted with a SupervisedMapper or a support vector machine to form groups that match the categories best.

2.5. Problems Related to Sequence Learning

HTMs, during training, rely on the temporal structure of the data. Even so, they partially throw this information away once the temporal groups are built. They perform their classification task on single frames of data only, ignoring the temporal context of the data. This is obviously harmful to any kind of sequence classification where most of the semantics is due to the order.

So if we, in spite of that, want to classify sequences with the help of HTMs, we have to transform the problem to a "regular" classification task. We could, for example, move a fixed-sized window over the sequence and at every step feed all the frames within into the network. That way, a part of the sequential context can be used to perform a purely spatial classification task. This, however, has several drawbacks:

1. The window size imposes some restrictions upon the length of recognizable sequences. Problems arise if sequences are so small that more than one fit into a window or if they are so long that the window does not cover them completely.

2. The size of the window is a parameter that has to be chosen by hand and remains fixed.

3. Because of the window-shifting over the sequence, all the bottom-layer nodes perform basically the same task. This implies a lot of overhead due to duplication.

4. Information collected for one step is lost for the next one. Hence, classification for the bottom level is redone several times as well as the disambiguation of input data.

5. As the window is shifted from one sequence to another, it will cover both of them for some time, thus leading to a blurred transition.

These problems partially arise because none of the nodes maintains memory of its history during inference. Also the temporal pooling ignores the order of its elements and therefore possibly merges different sequences. We will need to overcome these limitations in order to do successful sequence classification.

Another possibility would be to run a preprocessing which extracts temporal features from the data. The success of this approach is highly dependent on the type of features used. Therefore domain-specific knowledge is needed limiting the scope of this method.

3. State of Art

The following sections give a comprehensive overview of related work that considers Hierarchical Temporal Memory. Also a brief overview of alternative well known sequence learning algorithms is given.

3.1. HTM Related Work

Although the interest in HTMs is growing, only a few publications utilize them for sequence learning. This might be due to the problems pointed out in section 2.5 and the fact that all available implementations[1] by default only learn to classify spatial information. Also Numenta recommends not to use their framework for any kind of problem where "specific timing" [Numenta, 2007c] is required. Nevertheless, as mentioned in section 2.5, there are different approaches to still use HTMs for sequence classification tasks. Some kind of windowing or feature extraction could be used to turn the temporal information into a spatial representation that can be worked on with HTMs. Alternatively, the algorithms could be modified to fit the problem. All those approaches have been tried. There are three publications that utilize the first two techniques to overcome the limitations of the Numenta framework:

Firstly, Schey [2008] uses HTMs for song classification. Their training data consist of 5 different MIDI songs. For testing, every song is cut into approximately 20 pieces. Those excerpts are to be mapped back to their source by the network. Two approaches are taken to achieve good classification accuracy. The first one tries to classify the slightly pre-processed data directly without using any windowing or special feature extraction techniques while the second one utilizes both. As a result, the first approach fails obtaining less than 50% of accuracy while the second one works perfectly. However, this is not particularly surprising since there is no noise involved and the testing data is part of the training data. Therefore, the network does not have to do any abstraction and there is no guarantee against overfitting.

The technique of windowing is also applied by Sassi et al. [2009] who are using HTMs to boost the classification performance of SVMs in the field of motion capture data. A single device that captures accelerations is attached to the chest of a human. The resulting three-dimensional (one dimension for every axis) data is then used to tell apart different common movements like "standing", "walking", "jumping" and "falling". They use a 2-layer HTM network which receives a three-seconds window of the data that slides over the sequence. On top of the network, a Support Vector Machine is trained to do the actual

[1] There is the Numenta Platform for Intelligent computing available at http://www.numenta.com and also an open-source project called Neocortex available at http://sourceforge.net/projects/neocortex/.

classification. Thus the HTM essentially acts as a pre-processor for feature transformation. Unlike to the previous work, this one examines in detail the effects of white noise to the classification accuracy and HTMs are proved to be slightly improving the performance of a SVM in the case of noisy data.

The work of van Doremalen and Boves [2008] utilizes a set of features calculated by the Auditory Toolbox [Slaney, 1993] that encode the temporal information. That way, they were able to have an HTM produce reasonable results on spoken digit recognition.

While these three cited publications modified the data (feature extraction) or the way the data was read (windowing), there are also two groups that directly changed the HTM algorithms to better suit the task of sequence classification. The first one is Rozado et al. [2010] which studies the capabilities of HTMs to recognize signs from the Australian sign language. The different gestures were recorded using a data-glove that records both, the position of the hands as well as the position of the fingers. However, many of those signs share the same or at least very similar positions and the major difference between them is encoded in the order. To capture this temporal structure, they placed a modified node on top of their network. The behavior of this node is changed to store sequences during training instead of building temporal groups that disregard order. To decide which sequence is active, a variant of the Levenshtein distance is used as a metric. This modification basically utilizes an ordinary HTM network as a means to recognize the different poses, processing sequential information only on top of that. However for the case of the Australian sign language, this approach worked out quite well, yielding accuracies of over 90 %.

Maxwell et al. gave the algorithms a more general overhaul. By now, they have two publications [Maxwell et al., 2009a,b] and a third one [Maxwell et al., 2010] to come, dealing with their *Hierarchical Sequential Memory for Music (HSMM).* Although the focus of their system is sequence generation, in contrast to classification, the work is of particular interest for this thesis. Since most of the learning in HTMs happens in an unsupervised fashion, modeling the training data in a way that allows generation is also half the work for any classification task[2].

The modifications applied to the HTM algorithms are substantial. Briefly summarized, the temporal pooler of each node is substituted with a so-called *sequencer* which is able to extract reoccurring sequences from the training data and to match them with perceived sequences. Thereby, it is, in contrast to the Numenta temporal pooler, strictly keeping the sequential information. The whole setup is also enhanced by adding a feedback mechanism, thus allowing the network to generate low-level patterns from more abstract patterns at the higher levels. Furthermore, the execution policy is changed so that a node is only activated upon change, in contrast to activating every node at every time-step.

The new system is trained with pitch, velocity and rhythmic information extracted from a set of classical music MIDI files. The HSMM thereby builds up a model of that kind of music enabling it to generate similar melodies. Note that the network does not do any classification. It just tries to complete fragmentary melodies or to generate new ones. Therefore evaluation is difficult. In this case the authors personally judged the quality of

[2]For more details on this, refer to section 6.5 on page 56.

the created sequences lacking objective criteria. Though, according to their expertise[3], the results are promising.

In summary, these are all steps towards sequence classification with HTMs. But as the results show, the field is still in its infancy. The major problem of HTMs with this subject is to disregard the temporal order during inference. This problem has hardly been tackled. The work of Maxwell et al. is outstanding in this regard, but does not provide any classification capabilities.

3.2. Hidden Markov Models

Hidden Markov Models (HMM) are a popular model for sequence classification. They are used as a basis for many commercial speech recognition systems and have been successfully applied to problems from the sector of bio-informatics, visual recognition, finance and many others. Only recently have they been replaced by Conditional Random Fields in some of those applications.

A hidden Markov model consists of a finite set of states. The system transitions from one state to another in a stochastic manner and upon entering any state, a symbol is emitted, also stochastically, according to a probability distribution that is dependent on the entered state. The states themselves are not observed, neither at training nor at testing time. They correspond to the desired class labels. The emitted symbols, on the other hand, are always observed. Common tasks related to HMMs involve *decoding*, i.e. which state-sequence is most likely given the observed symbols and therefore which sequence of class labels corresponds to the data. In the terminology of this thesis *decoding* would be *inference*. Another common task is *parameter estimation,* i.e. which state transition and symbol observation probabilities best match the training sequence. Again in terms of this thesis, this would correspond to training. For a more in-depth discussion of HMMs see Rabiner and Juang [1986].

While HMMs are proven effective for many real-world problems, they also have some limitations (compare Kadous [2002] pages 40ff.). For this thesis, the most important restriction is that they can deal only with a single random variable. Even worse, hidden Markov models make strong independence assumptions about the distribution of that variable. Therefore, HMMs are not easily applicable to complex data such as video for example. This problem is one of the main motivations for this thesis.

3.3. Conditional Random Fields

Conditional Random Fields(CRF) can be viewed as a generalization of logistic regression to sequentially structured data [Sutton and McCallum, 2007]. They are a *discriminative* instead of a *generative* model which means that they do not model the joint distribution $P(\mathbf{x}, \mathbf{y})$ of their input data $\mathbf{x}$ and the class labels $\mathbf{y}$ but instead they only model the conditional probability $P(\mathbf{x}|\mathbf{y})$. One important implication of this is, that a discriminative

[3]James B. Maxwell is a composer and Arne Eigenfeldt is a Doctor of Music

model does not include a model of $P(\mathbf{x})$. This is not needed for classification because $\mathbf{x}$ is given at both training and testing time. So CRFs need no assumptions about the input data at all and do only restrict the distribution of the class labels $\mathbf{y}$. In the case of Linear-chain CRFs, this assumption is that each y_i only depends on the corresponding data x_i and the past label y_{i-1}.

The major advantage of Conditional Random Fields in comparison to hidden Markov models is that they can be applied to a much broader range of data. This is due to two facts: One, CRFs do not make any assumptions about the distribution of the input data and two, they utilize a set of feature functions as an interface to the data. The number of feature functions is fixed, so the size or even the type of the input data can vary. This makes CRFs very versatile. They have been successfully applied to a wide range of sequence classification problems including bio-informatics, medicine and intrusion detection [Liu et al., 2005, Chieu et al., 2006, Gupta et al., 2007]. For a good introduction see Sutton and McCallum [2007] or Klinger and Tomanek [2007].

3.4. Neural Networks

Neural Networks are a well known technique for classification tasks in general [Zhang, 2002]. However, classic feed forward neural networks such as the Perceptron are stateless, and therefore they are not directly applicable to temporal classification tasks. However there is the class of Recurrent Neural Networks(RNN) that seeks to remedy this problem.

Yet, the training of RNNs is more difficult than of feed-forward neural networks but different algorithms have been developed to solve those problems. Unfortunately, they can usually only be applied to pre-segmented data in order to produce a series of class labels. Recently Graves et al. [2006] showed how to overcome these difficulties and to use RNNs for labeling of unsegmented sequences. They have shown their system to out-perform different setups of HMMs on the well-known TIMIT corpus[4]. A main advantage of this approach is that it combines the spatial classification capabilities of neural networks with a temporal component. Unfortunately, those algorithms are not easily available so it is intractable for us to set it up for comparison.

[4] http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC93S1

4. Hierarchical Learning

The main contribution of this work is to integrate a sequence learning-approach into the hierarchical structure used by HTMs. In this chapter, the main ideas that precede the implementation will be developed. We start by summarizing the constraints of our task. Next, the process of learning a hierarchical representation of the sequence is described and possible problems are identified. Subsequently, we work out the individual steps of that procedure. We close this chapter by summarizing the completed algorithm. The final implementation including all the little details is postponed to the next chapter.

4.1. Premises

This section summarizes the pre-conditions that go with the task of sequence classification, the ones needed by Hierarchical Temporal Memory and finally, the assumptions we make about the data.

4.1.1. Sequence Classification

Informally, the task of sequence classification we want to deal with is to partition the sequence into parts and to assign a category to each of them. Kadous [2002] called this "*strong temporal classification*". Considering speech recognition, the parts could be phonemes, syllables, morphemes, words or even whole sentences depending on the desired granularity of classification. In the context of text processing, a common task would be to decide for each word (or group of words), whether it represents the name of a person, an institution, a place or something else. There are many other examples reaching from biology [Liu et al., 2005] to intrusion detection [Gupta et al., 2007].

To formalize the problem, we use the observation that partitioning the data and assigning a category to each block is the same as assigning a category to every symbol of the sequence. The partitions are then built indirectly as continuous sub-sequences sharing the same category label. So for a sequence $\mathbf{s}$ with elements $s_i \in \Sigma$ and category labels $c_j \in \mathbb{N}$ we can therefore define a sequence classifier as a function f which maps every given sequence over an alphabet Σ to a category sequence of the same length. Thus:

$$f : \Sigma^* \rightarrow \mathbb{N}^*, \text{ with } |\mathbf{s}| = |f(\mathbf{s})|$$

Resulting in for example:

$$\begin{array}{rcl} \mathbf{s} & = & s_1 \; s_2 \; s_3 \quad s_4 \; s_5 \quad s_6 \; s_7 \; s_8 \; s_9 \quad \ldots \\ f(\mathbf{s}) & = & \underbrace{c_1 \; c_1 \; c_1} \quad \underbrace{c_2 \; c_2} \quad \underbrace{c_3 \; c_3 \; c_3 \; c_3} \quad \ldots \end{array}$$

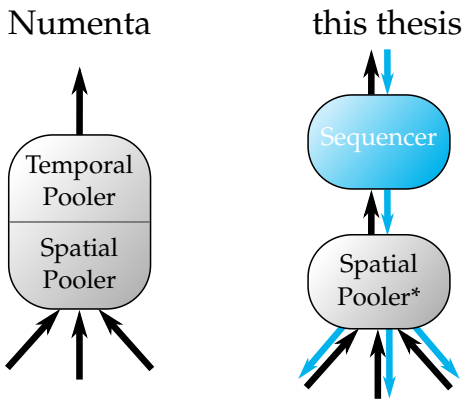


Figure 4.1.: Comparison of the structure of a Numenta node and the one we develop in this thesis.

Note that we cannot define the classifier as an *open-looped* one, i.e. one that outputs a category every time it receives a new symbol. If we did, we would force it to name the appropriate category as soon as it sees the first symbol. Thinking of an example like speech recognition, this would mean to know the correct word right after hearing the very first sound. This is obviously much more difficult than assigning a category afterwards.

4.1.2. HTM Framework

We want to maintain the ability of HTMs to build up a spatial hierarchical model of the training data. Hence, we keep the spatial pooler, the hierarchical topology of the network and the layer-wise bottom-up manner of training. Regarding the changes, we split the nodes, the temporal pooler will be replaced and we add a feedback mechanism.

The spatial pooling algorithm together with the hierarchical structure of the network are crucial for the learning process. They are used to mirror the spatial hierarchy in the data and model it accordingly. We also keep the general infrastructure. That is, input data is fed to the bottom layer nodes which perform their computation and pass the results upwards to their parents and so on. The unsupervised bottom-up manner of training is tied to the spatial pooler and the infrastructure. Thus, we adopt the Numenta fashion of training, too. Finally, we take over the idea of a common algorithm for all nodes[1]. Thus, all the nodes will perform both, spatial and temporal learning tasks.

These concepts we take over build a framework for all the changes we discuss in this chapter. It will be helpful to keep this in mind while keep on reading. Figure 4.1 highlights the modifications that will be done. We can see the main change is a substitution of the temporal pooler with a new component called sequencer. We also decided to split the node up. Furthermore, we add a feedback mechanism and (because of that) *slightly* adjust the behavior of the spatial pooler.

[1] This idea plays a major role in the Memory Prediction Framework, which is the foundation of Hierarchical Temporal Memory. However, the Numenta implementation is flawed in this respect. In the supervised case, the top-node performs a separate task. Furthermore does the behavior of the bottom spatial poolers differ from those of the others. Nonetheless will we adopt this idea.

4.1.3. Assumptions on Data Structure

Both, the sequence learning task and the HTM framework make assumptions about the structure of the data. Therefore, as we plan to combine both of them, we implicitly make those, too. Obviously the most important one is, the data having sequential nature with semantics inherent to the order of the frames within that sequence. Single sounds, out of any spoken language for example, carry only very few information about the text that is articulated. Even if all the sounds are given in a random order, they are nearly useless. Therefore it is clear that a major part of the semantics of speech is encoded through the order of the data frames.

Two more complex assumptions are made by the HTM framework. These are the hierarchical structure of both, space and time. Roughly summarized, this means that the patterns within the problem space can be split into a hierarchy of "sub-patterns". Analogously for time, the sequences consist of a hierarchy of sub-sequences. The individual parts are characterized by an increased auto correlation. Both concepts are explained in detail by Hawkins and George [2006]. Although undocumented, there is another assumption Numenta's approach requires: Considering the random Variable X^t representing the input data frame at time t and the hidden variable Y^t representing the corresponding pattern (or coincidence to use Numenta's nomenclature) the following holds:

$$P(Y^t|Y^{t-1}, Y^{t-2}, \ldots, Y^1, X^t, X^{t-1}, X^{t-2}, \ldots, X^1) = Pr(Y^t|Y^{t-1}, Y^{t-2}, \ldots, Y^1, X^t).$$

This means that the current coincidence depends only on the history of coincidences and the current frame but not directly on previous data frames.

Let us further assume that the borders of the desired category partitions coincide with the borders of some of the hierarchy components. So consider again the language example split into words and syllables. Then, the classes to learn would refer to groups of words and/or syllables but not to anything else. So for example classifying double-letters would not be feasible while part of speech classification would. We believe that this is a reasonable assumption and it will help us to use the hierarchical structure to perform classification.

4.2. Building up a Temporal Hierarchy

Let us now investigate the process of building a temporal hierarchy[2]. As we train in a bottom-up fashion, we start with the input data, split into frames along the time axis. We then group those into sequences that are then recursively grouped into sequences of sequences and so on. A schematic overview including the most important terminology can be seen in 4.2. To start in a simple way, we consider only one-dimensional data. Later on, we will extend that concept to higher dimensional data.

[2]What we mean by temporal hierarchy is really a hierarchical representation of the temporal structure of the data. But for the sake of readability, we will simply refer to it as temporal hierarchy.

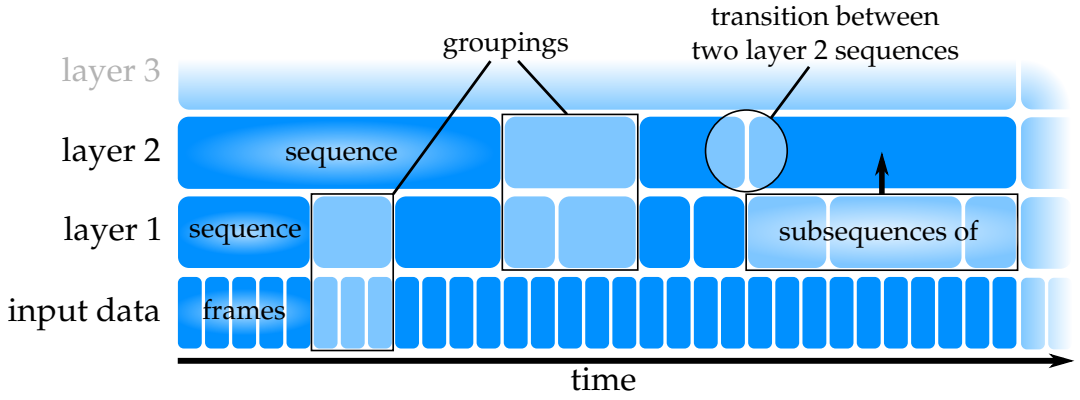


Figure 4.2.: Overview of the building process of a temporal hierarchy and the associated notation.

Considering for example speech recognition, would mean to group the audio-signal into phonemes that can then be grouped into syllables, words, sentences, and so forth.

The period of time represented by those groups grows as we ascend within the hierarchy and thus those groups also change less frequently. Consider for example the word "Singapore": The phonemes change quickly as it has nine different letters. But there are only three syllables and it is only one word. So the highest level concept does not change at all while the concepts of the lower levels may change quickly. Numenta calls this process "*coalescing of time*", Hawkins and George [2006].

Note that this procedure can only produce an efficient representation of the data if the shorter sequences can often be reused in different contexts. If we built a hierarchy on top of pure random data, we would find every possible sequence of a given length. Also, the next layer could find every possible combination of those sequences within the data and so on. Saving all of them is not more effective than storing the sequence as a whole. But if the assumption of a temporal hierarchical structure of the data holds, we will find only a few different sequences per layer that occur over and over again. This way we are able to reduce the complexity of the data at every layer of abstraction. This can furthermore improve the ability for generalization. For more details refer to Hawkins and George [2006].

To make the finished temporal hierarchy effective, a few conditions must be met:

1. *The grouping must be (easily) computable.*
   This is an obvious restriction since every node must be able to complete its task. Otherwise the whole procedure fails.

2. *On average, each group has to consist of more than one frame/sub-sequence.*
   This ensures that a coalescing of time takes place.

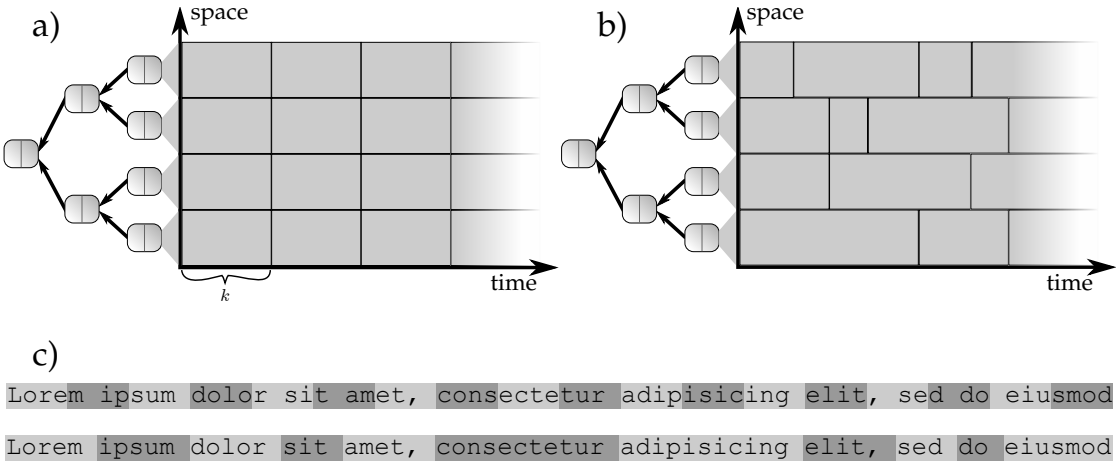


Figure 4.3.: Schematic visualization of the division of time in equal-sized partitions (a), dynamically depending on the data (b) and with a concrete text example (c).

3. *Groups should not "hide" higher level transitions.*
   This is a guideline to avoid inefficient representation at the higher layers. Consider two frames that get grouped together even though they belong to different groups at some higher level of abstraction in the hierarchical structure of the data. Then the groups of that layer cannot be built properly because the necessary transition has been "hidden" by the lower layers.

4. *Remain invariant against temporal translation.*
   Invariance with respect to temporal translation is a property of almost every time-dependent task. It simply means that the absolute starting time of a sequence does not affect its semantics. Note that we only refer to the absolute position in time, not to relative translations of sequences as this obviously can change the meaning, at least for some higher level concept.

To sum up, we have explained the concept of a temporal hierarchy and discussed some of the pitfalls when trying to build one.

4.3. Temporal Grouping

From the perspective of a single node, the problem reduces to a stream of *coincidences*[3] which is to be grouped into sequences according to the guidelines given in the last section. We refer to this task as *temporal grouping*. In this section, we will study how this task can be solved.

[3]Coincidences are found by the spatial pooler. Each coincidence can either represent a frame or a sub-sequence found by a child node. But in the end this distinction doesn't matter for the task the node has to perform.

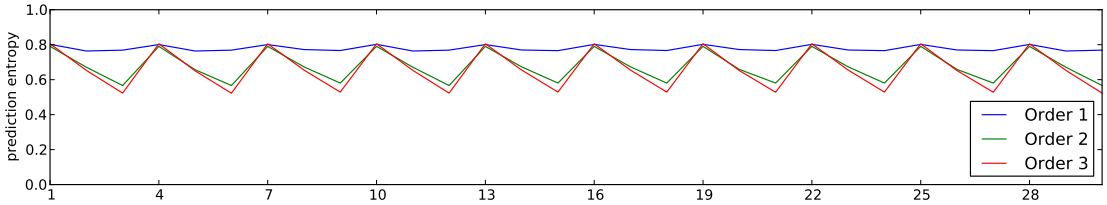


Figure 4.4.: Average prediction entropy using a k-th order Markov Model for a sequence composed of three different sub-sequences over an alphabet size of 3. The average is formed by all possible sub-sequences.

The first and most simple idea we could think of is to pack the coincidences into equally sized sequences (compare figure 4.3a). That way, the grouping would be easily computable and we can guarantee that on average each group consists of more than one coincidence. So the first two conditions from section 4.2 are met. On the other hand, transitions between blocks are almost randomly hidden (such as for example the word transitions in figure 4.3c). Even worse, it is not invariant with respect to temporal translations. So this procedure is not an option at all. It also becomes clear that any data-independent partitioning will suffer from these two problems. Hence, we need to adjust the grouping dynamically to the actual data (see figure 4.3b for illustration).

To solve these problems, we adopt the concept of statistical segmentation of Golcher [2006]. Roughly, the idea is to suspect a group transition every time the predictability of the next coincidence is low. To understand this approach, we have to study the impact the temporal hierarchical structure has upon the predictability.

We have already seen, that small sequences that occur often in different contexts are building blocks of such a hierarchy. That in mind, we will consider the conditional probability for a symbol given a small context of k symbols before. That is a k-th order *Markov Model* used for prediction of the next symbol. What we notice is that on average the prediction is much stronger for a symbol in the end of a block than for a symbol at the beginning. Intuitively this is clearn because each block occurs several times. If we are therefore able to recognize such a block using the k-symbol context, then guessing the next coincidence will be easy until the end of that sequence. Once it ends though, we might have trouble predicting the next sequence being limited to such a small scope. We will not prove this correlation here but figure 4.4 shows some empirical evidence. We can see that the entropy is the highest for the first symbol of every block. Also the difference increases with the order of the Markov model.

As a measure of "predictability", consider the entropy H of a discrete random variable Y with possible values $\{y_1, y_2, \ldots, y_n\}$:

$$H(Y) := \sum_{i=1}^{n} P(y_i) * \log_2(P(y_i)).$$

Entropy is commonly used as a measure of uncertainty. We can normalize it by using the logarithm with base n instead of 2, thus making it reach from 0 to 1. The closer

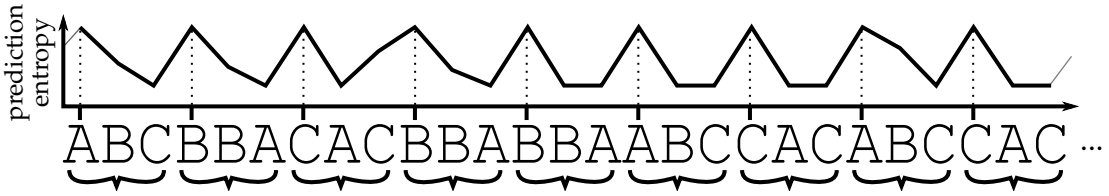


Figure 4.5.: The prediction-entropy for an artificial data set consisting of the letters `A`,`B`, and `C`. Those are arranged in a hierarchical way to appear only in the groups `ABC`, `BBA`, and `CAC`. The plot shows the entropy of a prediction with a context length of 3 and the entropy-induced separation in groups.

to 1 the entropy is the less information we have about the next coincidence. An entropy of 1 corresponds to the equal distribution over all possible values. On the other hand, the more the entropy approaches 0, the stronger is the prediction, reaching 0 only if the probability for any symbol is exactly 1.

Following our thoughts from above, we conclude that a coincidence with a high entropy is likely to be the onset of a new group. Therefore, it is a reasonable choice to partition the input data according to the entropy, starting a new group every time the entropy passes a certain threshold. The coincidence whose prediction-entropy was high becomes the first symbol in the new group (compare figure 4.5). Note that the order of the Markov-predictor does not correspond to the length of the groups. The groups may be much longer than the order if the symbols within are easily predictable using only a context of k symbols. Likewise shorter sequences are possible.

4.4. Prediction

As we have seen, prediction plays an important role for the process of temporal grouping. This section explains which assumptions are needed to make appropriate predictions, what algorithms we use to calculate them, how to deal with uncertainties in the input data and, finally, how they can be used in addition to the temporal grouping.

4.4.1. Assumptions

The most important question when dealing with predictions is which assumptions to make. Because without assumptions, no future values can be foretold. In our case two assumptions are made: First, the sequence is discrete taking only values that also appeared in the training-data. Second, as we have seen, some context is needed in order to make useful predictions for the process of temporal grouping. Thus, we assume the Markov property for a small order of $2 \leq k \in \mathbb{N}$ with k as a node-parameter. This means that the value of the random variable Y^{t+1} at time-step $t+1$ only depends on the k values before. Thus:

$$P(Y^{t+1}|Y^t, Y^{t-1}, \ldots, Y^1) = Pr(Y^{t+1}|Y^t, Y^{t-1}, \ldots, Y^{t-k+1}).$$

Note that none of these are new assumptions. They all derive from those discussed in section 4.1.3 on page 14. The data is discrete and takes only known values because the spatial pooler converts the sequential data into a sequence of coincidences. And although the Markov property does not necessarily hold for the whole data, it certainly does within the blocks of our hierarchy. In fact, one could say that we even make use of the circumstance that it does not hold in between blocks to detect transitions.

4.4.2. Algorithm

The class of Markov models is well studied. It is therefore not surprising that there are many good prediction algorithms based on those assumptions. In particular *Variable Order Markov Models(*VOMM*)* have some prominent representatives in the field of lossless data compression. As shown in Feder and Merhav [2002], data compression has a tight relation to sequence prediction. Six of them are compared in Begleiter et al. [2004] with respect to their sequence-prediction capabilities: *Context Tree Weighting* (CTW), *Prediction by Partial Match* (PPM), *Probabilistic Suffix Trees* (PST), *Lempel Ziv 78* (LZ78), and an *Improved Lempel Ziv* (LZ-MS). Because of the very good results of PPM, we chose to utilize this algorithm for prediction.

4.4.3. Dealing with Uncertainties

The mentioned algorithms all operate on discrete sequences. Hence, they have no built-in ability to deal with uncertainties in the input data. But the data received from the spatial pooler is a probability distribution over coincidences and not a discrete sequence. So we must find a way for our Prediction algorithm to deal with this fuzzy data. The correct way would be to calculate a prediction for *every possible context* and average them weighted with the probability for the associated context. Let us formalize this:

Let $\mathcal{C} = \{1, 2, \ldots, |\mathcal{C}|\}$ be the set of possible coincidences recognized by the spatial pooler. We can then define the context $\mathbf{c}$ for the prediction as a k-tuple of coincidences: $\mathbf{c} = (c_0, c_1, \ldots, c_{k-1})$ with $c_i \in \mathcal{C}$ thus $\mathbf{c} \in \mathcal{C}^k$. Let X^t be the random variable representing the pattern at time step t. Let further $p_i^t = P(X^t = i)$ be the probability, calculated by the spatial pooler that the coincidence at time step t is i. The probability for any context $\mathbf{c}$ at time $t > k$ follows as:

$$P(\mathbf{c}) = \prod_{i=0}^{k-1} p_{c_i}^{t-i}.$$

The prediction function $f : \mathcal{C}^k \times \mathcal{C} \to [0, 1]$ provided by the PPM algorithm maps a probability to every combination of context and pattern. Thus:

$$f(\mathbf{c}, i) = \hat{P}(X^{t+1} = i | (X^{t-k+1}, X^{t-k+2}, \ldots, X^t) = \mathbf{c}).$$

Of course $\sum_{d \in \mathcal{C}} f(\mathbf{c}, d) = 1$ must always hold. The above described prediction for any coincidence $d \in \mathcal{C}$ under uncertainty follows as:

$$\hat{P}(X^{t+1} = d) = \sum_{\mathbf{c} \in \mathcal{C}^k} \left[P(\mathbf{c}) \cdot f(\mathbf{c}, d) \right]. \tag{4.1}$$

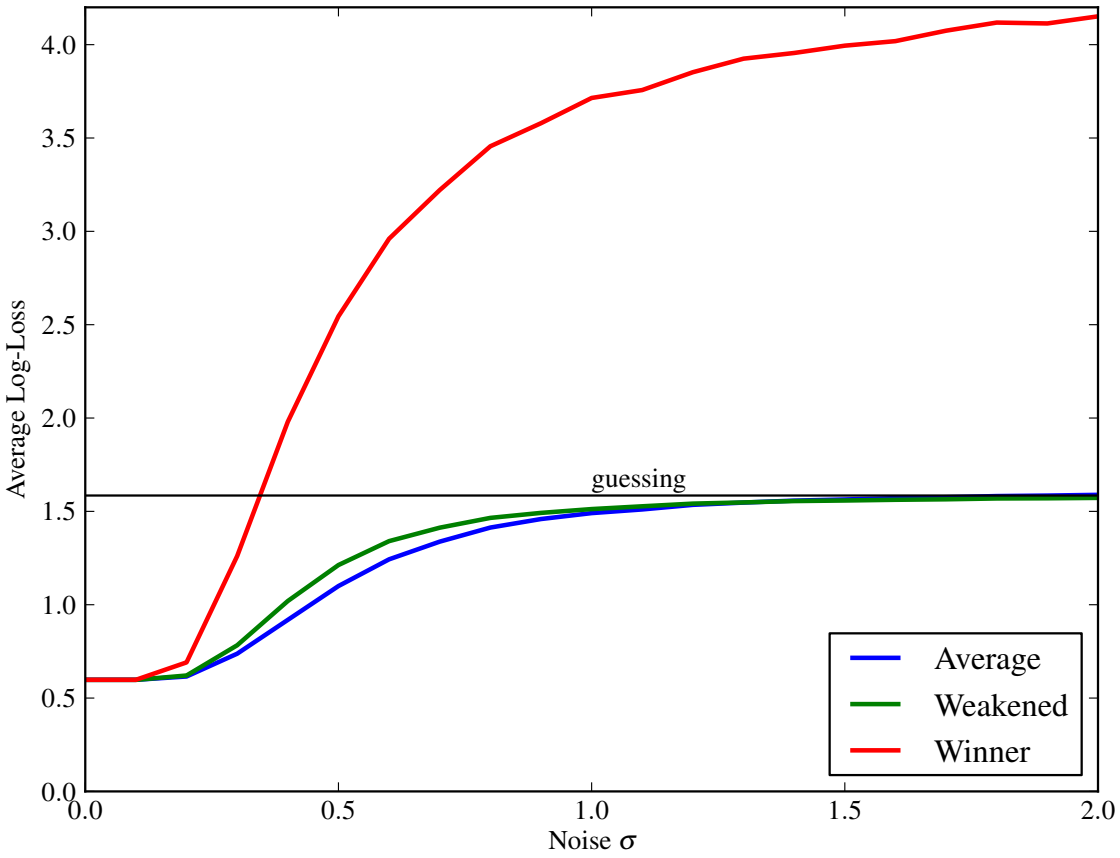


Figure 4.6.: Comparison of the different prediction-approaches under uncertainty. The "Average" line shows the results for equation 4.1, "Weakened" corresponds to equation 4.3 and "Winner" shows the results for equation 4.2.

Now, at the latest, it becomes clear that this approach is intractable because $|\mathcal{C}^k| = |\mathcal{C}|^k$. Thus, the number of calculations quickly becomes too big to be manageable. At the other extreme we could just ignore all the uncertainties and use only the most likely context $\mathbf{c}^*$:

$$\mathbf{c}^* = \arg\max_{\mathbf{c}\in\mathcal{C}^k} \prod_{i=0}^{k-1} p_{c_i}^{t-i} = (\arg\max_{c_0\in\mathcal{C}} p_{c_0}^t, \arg\max_{c_1\in\mathcal{C}} p_{c_1}^{t-1}, \ldots, \arg\max_{c_{k-1}\in\mathcal{C}} p_{c_{k-1}}^{t-k+1}).$$

And thus:

$$\hat{P}(X^{t+1} = d) = f(\mathbf{c}^*, d). \tag{4.2}$$

The calculations become very easy with this approach. Yet, experiments show that by doing so, the predictions quickly turn very bad in the case of noisy data. The reason is that any small error that changes the context might yield a completely wrong prediction.

A compromise between the two is to average the most likely context with the equal distribution, thus weakening its effect on the input data. This means calculating the final prediction as follows:

$$\hat{P}(X^{t+1} = d) = P(\mathbf{c}^*) \cdot f(\mathbf{c}^*, d) + (1 - P(\mathbf{c}^*)) \cdot \frac{1}{|\mathcal{C}|}. \tag{4.3}$$

Although not as exact as equation 4.1, this approach proves to be robust against noise while still being easy to calculate. This strategy can easily be generalized to work with an arbitrary number of different contexts taken into account. That way the trade-off between precision and calculation effort can be adjusted smoothly. Consider a sub-set of all contexts $\mathcal{C}^* \subseteq \mathcal{C}^k$. Then, analogous to equations 4.1 and 4.3, we can define:

$$\hat{P}(X^{t+1} = d) = \sum_{\mathbf{c} \in \mathcal{C}^*} [P(\mathbf{c}) \cdot f(\mathbf{c}, d)] + (1 - \sum_{\mathbf{c} \in \mathcal{C}^*} P(\mathbf{c})) \cdot \frac{1}{|\mathcal{C}|}. \tag{4.4}$$

The effects of the different prediction approaches are shown in figure 4.6 on the preceding page. As we can see, all three start with an average log-loss of ≈ 0.6. But as the level of noise increases, they become worse. The "Average" line stays consistently the best while the "Winner"-line even exceeds the value of ≈ 1.6 which would result from pure guessing.

4.4.4. Feedback for the Spatial Pooler

So the node will try to predict the next pattern at every time step. This information can be used to help the spatial pooler disambiguate its input in case of noisy data. This can simply be done by multiplying the prediction with the probability distribution the spatial pooler has calculated. The result is a combined belief using both, the spatial and the temporal structure of the data.

4.4.5. Prediction as a Measure of Success

Using a prediction-based temporal pooling algorithm has another advantage. Since we cannot calculate a classification accuracy in a unsupervised context, we essentially lack a method of measuring the performance of a single node. But since it makes predictions all of the time, we can measure its success in modeling the input data. This will help us to study the impact of modifications and the effect of feedback.

4.5. Feed-Forward Data

In this section, we will work out the details for the feed-forward data the new node sends to its parent. We have already seen, how the node can group its input data into sequences. Now we want to find a way to encode these information into some vector the next layer can use to repeat the sequencing task. So let us start by summarizing the goals:

1. The feed-forward data must be a vector of real numbers. This ensures that it can be used by the spatial pooler of the next layer.

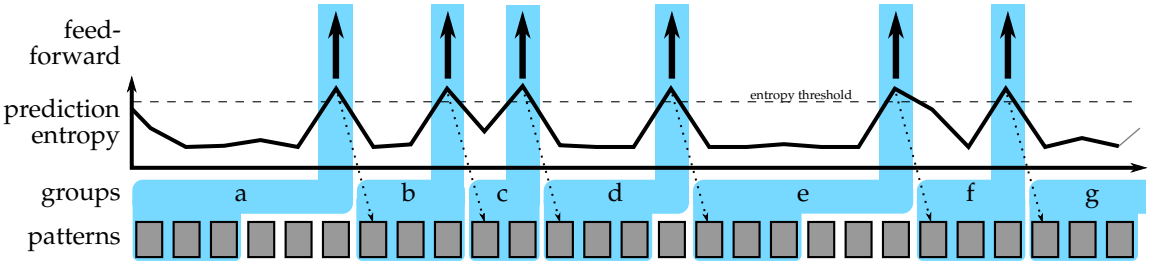


Figure 4.7.: Feed-forward data example with a Markov-order bound of 3. We can see a sequence of schematic patterns at the bottom formed into seven groups in the second line. Above that, there is a plot of the corresponding prediction-entropy and the entropy threshold. On top, we can see arrows indicating the timing of the feed-forward message. Notice that only a maximum of three patterns identify each group although it actually might be bigger.

2. It should be suited to discriminate all the sequences. That way, the next layer can build up a model of how to combine those into even bigger sequences.

3. Feed-forward data should only be sent once every sequence. Otherwise, the layer above cannot make proper use of the grouping because from its perspective every incoming data represents a frame/sub-sequence.

4. We want to be able to encode uncertainties.

We choose a representation, similar to the one the spatial pooler and the temporal pooler use to have one component for each group. The value of that component represents the degree of certainty the node has about this sequence being active. After normalization, it will be a probability distribution.

Since the groups can have common prefixes, it might be impossible to tell them apart if they have not yet finished. So in order to give an accurate distribution we have to wait for the very last time-step of a sequence to send information upwards. This moment can easily be recognized by calculating the entropy of the prediction for the next symbol. If this entropy is higher than the average, the current group ends instantly. This moment is still early enough to ensure that the parent node has the possibility to send some disambiguating feedback just in time. At that point, this kind of feedback is highly welcome because the prediction is not well suited to resolve ambiguities (it has a high entropy). An example for the feed-forward timing can be found in figure 4.7.

The final question we need to answer is: How should we compute the certainty of a sequence being active? They can differ in length which makes comparing them difficult. We choose to restrict calculation to the first k symbols of a sequence. The reason for this is simply that if two groups shared a common prefix of k symbols, they would be completely equal from the perspective of the predictor. So their predictive entropy would be exactly the same. This also means that they would not have been regarded distinct during grouping. Therefore, from the perspective of the sequencer, they are equal. So for our purpose those first k symbols suffice to distinguish sequences.

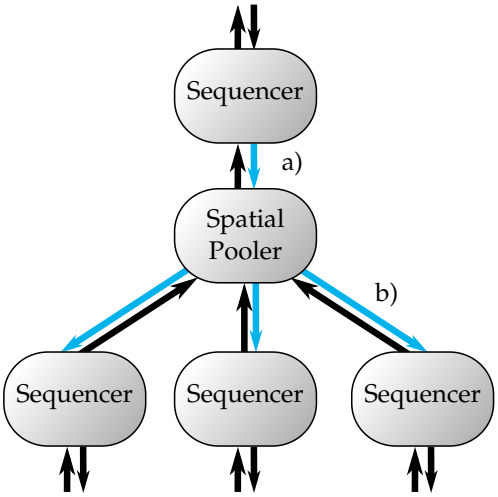


Figure 4.8.: Extract from a simple network to illustrate the two kinds of feedback. a) From a sequencer down to its spatial pooler child and b) from a spatial pooler to its children.

Hence, we can just multiply the probabilities for the first k symbols of the sequence to obtain a measure α of certainty for the sequence $\mathbf{s} = s_1 s_2 \ldots$:

$$\alpha(\mathbf{s}) = \prod_{i=1}^{k} p_{s_i}^{t-k+i}. \tag{4.5}$$

Recall that p_i^t denoted the probability received from the spatial pooler for coincidence i being active at time-step t. If the length of $\mathbf{s}$ is less than k, the missing coincidence probabilities are taken to be $\frac{1}{|\mathcal{C}|}$ with $\mathcal{C}$ the set of all coincidences. Finally, the complete feed forward data $\mathbf{x}^t$ at time t follows as a normalized vector of certainties over all sequences $\mathbf{s}^j \in \mathcal{S}$:

$$x_j^t = \frac{\alpha(\mathbf{s}^j)}{\sum_{i=1}^{|\mathcal{S}|} \alpha(\mathbf{s}^i)}.$$

4.6. Feedback

As mentioned in the beginning, we want to add a feedback mechanism. Taking a look at the structure of our network, we will discover two different types of feedback (compare figure 4.8).

4.6.1. Sequencer to Spatial Pooler

The first kind of feedback is basically the prediction the sequencer makes about the next frame. This prediction is a distribution over the patterns formed by the spatial pooler.

Therefore, they can be passed down unmodified. At the next time-step it will simply be multiplied with the distribution the spatial pooler calculated helping to resolve potential ambiguities.

But we can do even better than that. Every time the sequencer passes some data upwards, its prediction will be weak. We know this because it only passes data upwards if the entropy of the prediction is high. But we also receive feedback $\boldsymbol{\pi}$ (see figure B.1 on page 78 for an overview of the notation) from the parent every time we pass data upwards. This feedback will essentially be some prediction about the anticipated sequence starting at the next time-step. So we can translate this into a potentially stronger prediction $\boldsymbol{\psi}$ about the next coincidence by using the first symbols $s_{j,1}$ of all sequences weighted by the probability π_j for the sequence[4] :

$$\psi_i = \sum_{j=0}^{|\mathcal{S}|} \mathbf{1}_i(s_{j,1}) \cdot \pi_j.$$

With:

$$\mathbf{1}_i(j) = \begin{cases} 1 & \text{, if } i = j \\ 0 & \text{, otherwise} \end{cases}.$$

So we have two cases calculating the feedback for the spatial pooler. If the entropy of the prediction is lower than average, then the prediction is the feedback. Otherwise the sequencer waits for the feedback of its parent and transforms it into a distribution over coincidences. That way, we also ensure that additional layers will have influence on the disambiguation at the bottom layer. Note, however, that their impact will be smaller the further they are away from the bottom layer. This is because for most symbols the first layer will make the predictions. Only occasionally, if it fails to make a good prediction, the second layer will step in. If the second layer is to provide the prediction and it cannot make a strong one, only then will the third layer help, and so forth. Informally one could say that the layer above only helps out in case of doubt.

4.6.2. Spatial Pooler to Sequencers

The spatial pooler does not create feedback on its own. Its task is to translate the feedback $\boldsymbol{\psi}$ it receives (and uses) from the sequencer into feedback $\boldsymbol{\pi}$ for its children. This procedure is pretty straightforward:

$$\boldsymbol{\pi} = \sum_{i=1}^{|\mathcal{C}|} \mathbf{c}_i \cdot \psi_i.$$

Notice that the coincidences $\mathbf{c}_i$ are vectors, so the summation is a vector addition. Finally, the resulting $\boldsymbol{\pi}$ has to be normalized and split up into appropriate chunks for every child.

[4] Of course the result needs to be normalized.

4.7. Unfolding

We add one more concept that we will call *unfolding*. The idea is to reverse the abstraction process done by the network and to project sequences found at the top-level back to the input-data it would correspond to. So every time the top node passes a distribution upwards, we send it back through the network and try to reconstruct the input-data sequence it corresponds to. Because we will only use the stored sequences and coincidences, this will not match the real input data in general. But it will reflect the internal model of the input data the network has. If we consider the speech recognition example again, unfolding would turn the concept of a word (which would have an ID) back into a stream of sounds that corresponds to that word.

Therefore, unfolding can help us to understand what happens within the network. Furthermore, it will also allow us to perform some kind of reconstruction of missing or noisy input data because the internal model of the network is always "clean" and complete. So by constructing an input sequence according to that model, we will remove noise and restore missing values. However, this procedure will only remove what the network thinks of as noise. We can also use this approach as an alternative way of classifying: by "reconstructing" missing class labels. We will discuss this in detail in section 6.5.

Now, we will present how unfolding works: Every node unfolds its kind of *patterns* that is a sequence in case of a sequencer and a coincidence in case of a spatial pooler. The result is a set of channels where every channel corresponds to one bottom-layer node. So every channel contains a sequence of input-frames for that node.

For the case of spatial poolers, the unfolding of a coincidence means to take that coincidence and split it up for all of its children. For each of the parts, the winning pattern is determined. This corresponds to the ID of the pattern the child is then instructed to unfold. Each of these yield a set of channels which are unioned. Together, they correspond to the descendant bottom-layer nodes.

Likewise, a sequencer node determines the coincidences that belong to the sequence to be unfolded. Then, it successively instructs its child to unfold each of them. The resulting channels are concatenated, so that the number of channels stays the same but each of them grows longer in time.

We want to compare the end result to the real input data. Therefore, we have to make sure that it is exactly the same length as the part of input data we want to compare it to. Unfortunately, this is not generally the case if we unfold a sequence in the described way. Length differences can occur for example due to temporal noise in the real data. While unfolding, we "reconstruct" a sequence with no noise and therefore might end up with a different duration. In order to enforce the correct length, every sequencer keeps a memory of all the time steps that it sent data upwards. That way we can tell it to unfold sequence number i starting at frame t and it will know how long it took in the original data and hence unfold it appropriately.

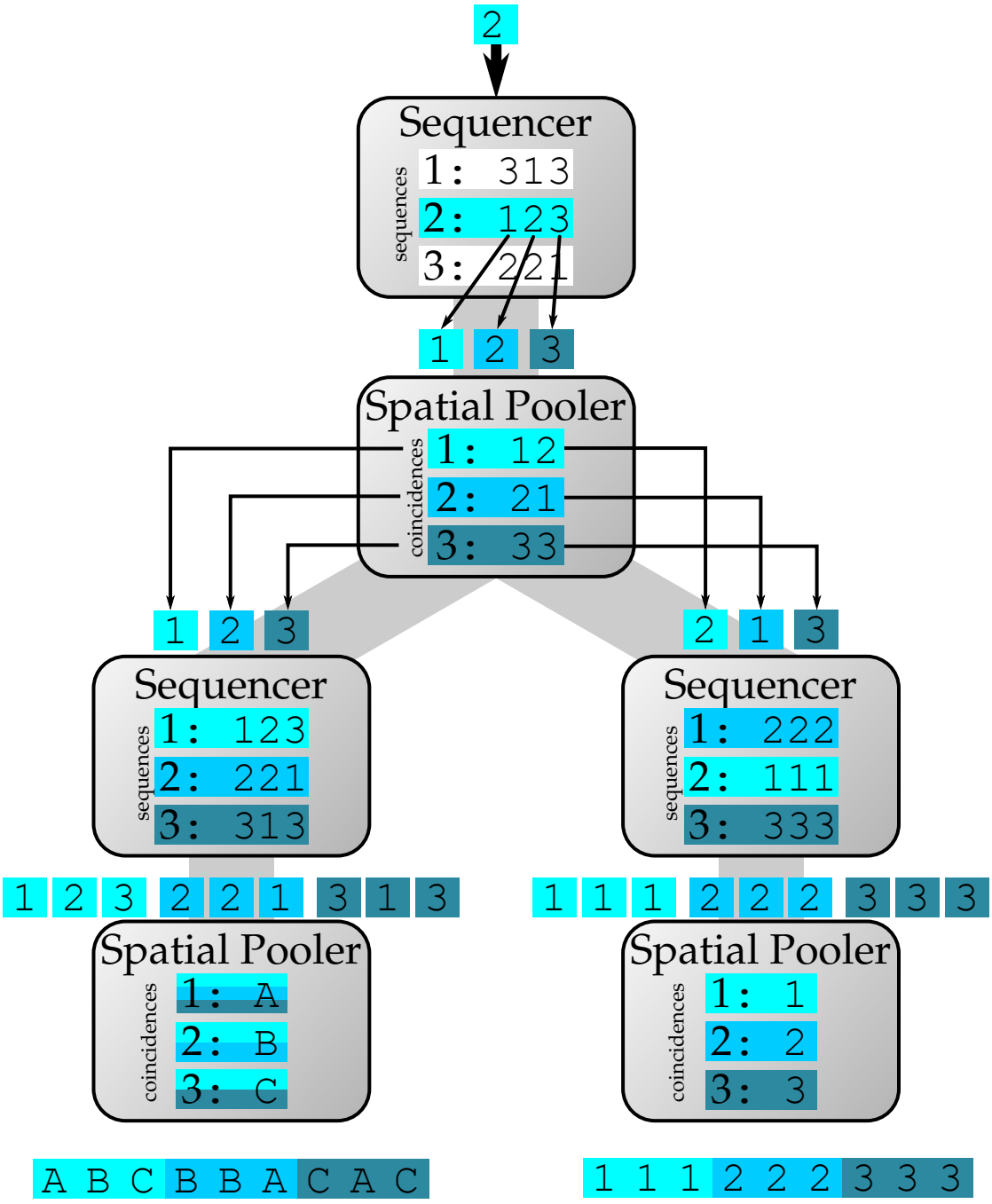


Figure 4.9.: Example for the unfolding process. We can see a network consisting of six nodes. The top sequencer is told to unfold sequence number 2. While the unfolding works its way down the hierarchy, we can see the sequence grow resulting in channels of length 9 each. These two are the result of the process. Colors and arrows help to keep track of the sources for the symbols.

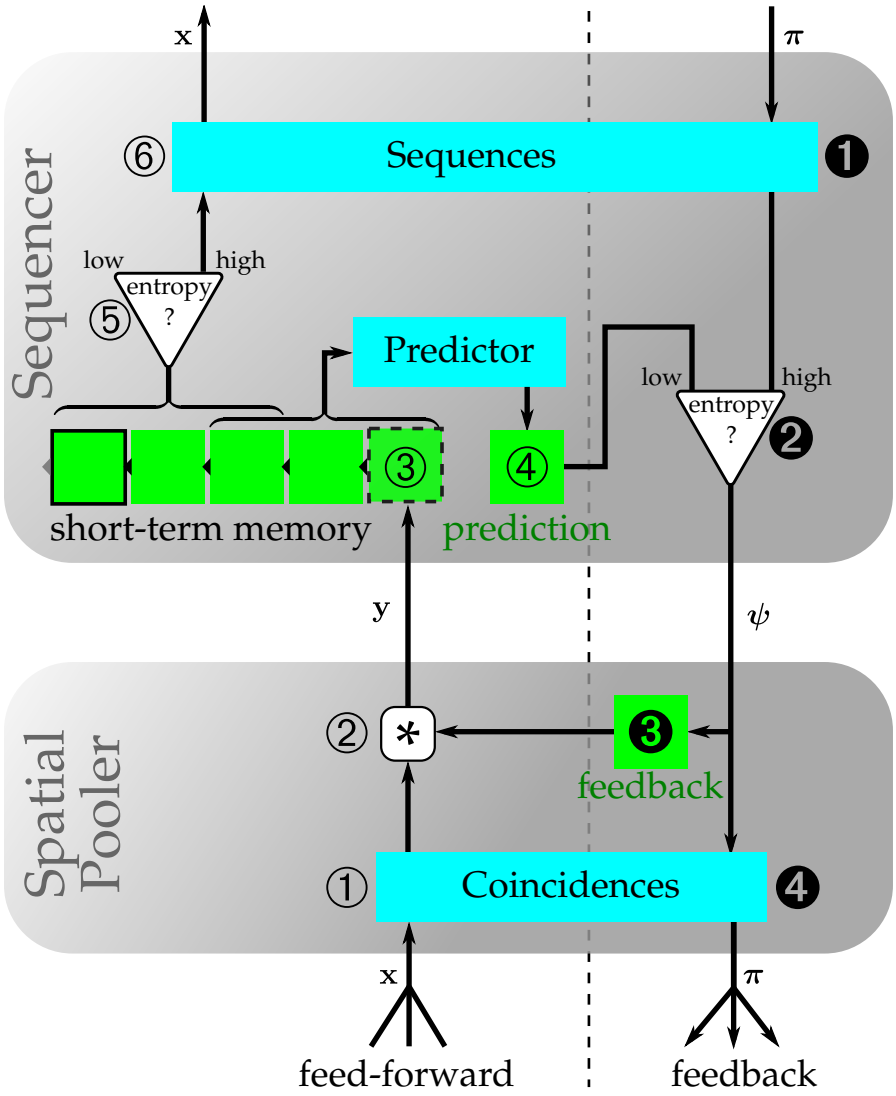


Figure 4.10.: Schematic overview of the operations of the modified spatial pooler node and the new sequencer node. The cyan colored boxes represent the knowledge acquired during training. They do not change during inference. On the contrary, the green colored boxes depict internal variables that are constantly updated.

4.8. Putting it All Together

We finish this chapter by summarizing the new node algorithm as a whole. We will often refer to figure 4.10 on the preceding page which illustrates the operations of both node types during inference. The individual steps within that diagram are referenced using the corresponding circled numbers (①②③④⑤⑥❶❷❸❹)

4.8.1. Spatial Pooler

The spatial pooler has only been added a feedback mechanism. Other than that, it behaves just like the Numenta one.

Training

During training, the node collects a list of coincidences. Every time the input data differs more than $maxDistance$ from any coincidence, it is added to the list. This is repeated until the number of coincidences reaches $maxCoincidences$ or the training is finished. So far, the algorithm matches exactly the old one.

Inference

During inference, the probability of the input data $\mathbf{x}$ to correspond each of the stored coincidences is calculated ①. This is done using a simple mixture of Gaussian model with means equal to the coincidences and a standard deviation of σ (the node-parameter). For details compare algorithm A.2 on page 69.

Before that distribution is passed upwards, it is multiplied with the feedback of the previous time-step (if any) and thereafter normalized again ②. This is sensible because the feedback received at the last time-step represents a prediction for the current time-step. The resulting distribution is then passed to the parent.

If feedback from the parent is received, two different things are done. First, it is saved for the next time-step to be multiplied ❸. Second, it is translated into feedback $\boldsymbol{\pi}$ for the children ❹. This means calculating an average of all coincidences weighted with the probabilities from the parental feedback $\boldsymbol{\psi}$. Note that $\boldsymbol{\pi}$ is a probability distribution over all coincidences while $\boldsymbol{\psi}$ is of the same type the input data was which is a probability distribution for all layers but the bottom-layer.

4.8.2. Sequencer

The sequencer node replaces the Numenta temporal pooler node. There is always one sequencer on top of every spatial pooler.

Training

The sequencer stores the whole stream of input data while it is training. Also for every received distribution, the winning symbol (coincidence) is determined and it is used to train a VOMM predictor. After the whole training-sequence has been received, the (now

trained) predictor is used to calculate a prediction and hence entropy for every frame. The whole stream is then partitioned into sequences according to that entropy. More precisely, a new sequence is started every time the entropy exceeds its average and it is truncated to k symbols. Finally, duplicates are removed as well as sequences that appeared too infrequently. So by the time the training has finished, the node has a trained VOMM predictor and a list of common sequences.

Inference

While inferring, the node appends each incoming $\mathbf{y}$ to its short-term memory ③. The short-term memory holds all the coincidence distributions since the beginning of the current sequence[5] but at least k.

After that, the winning coincidences for the most recent k distributions are determined. They are used as a context for the Predictor to predict the next symbol and store it in the prediction field ④. In addition to that, the entropy of that new prediction is calculated. If the entropy turns out to be higher than average[6], the first k symbols of the current sequence are taken from the short-term memory ⑤. Then they are used to calculate a distribution over all sequences ⑥. Recall that the certainty for each sequence is calculated as the product of the probabilities for its individual symbols (compare equation 4.5 on page 23). The resulting vector $\mathbf{x}$ is normalized and passed upwards as feed-forward data.

Every time feedback $\boldsymbol{\pi}$ is received from the parent node, it is transformed back into a distribution over coincidences ❶. This is done by finding the average of the first symbols of all sequences weighted with the probability for that sequence given by $\boldsymbol{\pi}$. Note that $\boldsymbol{\pi}$ is a distribution over all sequences while the resulting vector represents a distribution over coincidences. But this vector is only used as feedback $\boldsymbol{\psi}$ if the entropy of the "own prediction" is high. Otherwise, the own prediction is used for feedback ❷.

4.8.3. Execution Order

We need to make sure that the feedback from the higher layers arrives at the lower ones in time. But every layer waits until the current sequence has finished to pass information upwards. So in the worst case, we have the first layer pass data upwards to the second one which also passes data upwards to the next layer and so on all the way up to the top node. The top node, of course, cannot delegate the feedback creation at step ❷, and hence sends its prediction as feedback downwards. But the entropy of all sequencer nodes on the way down is high because otherwise, they would not have passed data upwards. Therefore, all the nodes will use the transformed feedback from above as feedback. So in summary, we have the input data travel all the way up through the hierarchy and the feedback travel all the way back down again, all in one frame. Thus, the execution order must be to first process the feed-forward data (steps ①②③④⑤⑥) of every node and only then process the feedback (steps ❶❷❸❹) of all the nodes in reverse order.

[5] Thereby each sequence begins the time-step after the last one has been passed upwards, that is every time the entropy of the prediction for the current symbol is higher than the average.

[6] This is the average calculated over the training data.

5. Implementation

This chapter gives a brief introduction to the SeQuence Hierarchical Temporal Memory (QHTM) implementation we have written to test the algorithms. It provides a basic framework for HTM-like computations and it is easy extensible. The software is available under the GPL3 license[1].

5.1. Basic Architecture

The software is entirely written in Java Edition 6. It relies on the Logback framework[2] for logging purposes. The configuration is realized with XML files that are read and written using the JDom[3] library. The graphical user interface is based on Swing and uses the additional SwingX[4] and dockingFrames[5] libraries for non-standard components. The code of Begleiter et al. [2004][6] for the Prediction by Partial Match algorithm is used by the Sequencer. The SVMTopNode uses the Java version of libSVM[7].

The QHTM software is designed to run a given hierarchical network of nodes under specified conditions. Therefore, the two main parts are:

1. The *network* which holds all the information about the nodes, their parameters, the sensors and the topology.

2. The *batch tree* which cares for the actual process of execution. Which data the network is trained and tested on, what the accuracy was and so forth.

This modularization allows a flexible combination of networks and tests. Both, the network and the batch tree are stored in separate XML files. The main application is only responsible to load them. All the further setup and the computations are then carried out by those two parts.

An important design decision for this software is to have all the features and configurations accessible via XML files in order to achieve fully featured batching capabilities. The XML files consist of a standard XML header and a root node corresponding to either the network or the root of the batch tree. Each component can contain child tags defining some of its parameters or sub-components. Thereby, the XML tree structure resembles the structure of the network respectively the batch tree.

[1] The source-code can be downloaded here: MISSING
[2] http://logback.qos.ch/
[3] http://www.jdom.org/
[4] http://swinglabs.org/
[5] http://dock.javaforge.com/
[6] http://www.cs.technion.ac.il/~ronbeg/vmm/
[7] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

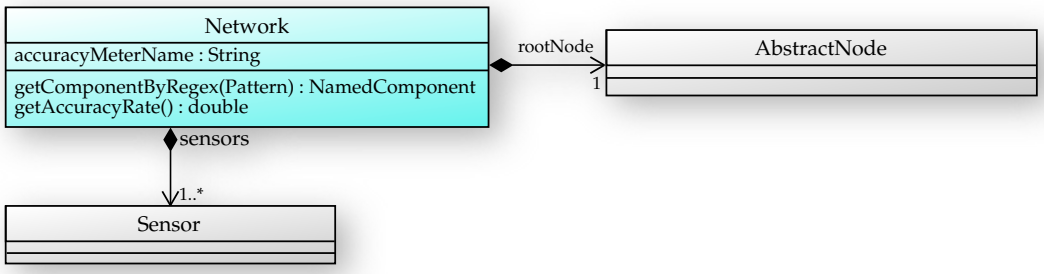


Figure 5.1.: Simplified part of the UML diagram showing the `Network` class.

The `Parameter` annotation is used for fields of any component that should be treated as parameters. Those can be set for example by the XML file or by the graphical user interface. Internally, the fields are accessed via reflection using the annotation as a marker to distinguish parameters from other fields. The static class `ParameterAccess` handles all the getting and setting of those parameters.

The main benefits of this approach are that the XML export and import of parameters can be automatized and that the GUI can easily provide a general interface to view and edit the parameters of any component without further customization.

An important goal was to create a flexible framework that could be easily extended and modified. Therefore, a general component conversion from and to XML as well as a GUI that is able to display every custom implemented component without further adaptions are crucial. To accomplish this, all the components are instantiated via reflection. This makes component development painless but imposes some restrictions on the construction of them. The most important one is that every component must have a parameter-free constructor. Thus most of the initialization must be separated from construction.

5.2. Network

This section summarizes the structure and the concepts of the network. Its only parameter is the so called *accuracyMeterName* which holds the name of the component that should measure the accuracy. This could be a node, a sensor or an effector as long as it implements the `AccuracyMeter` interface.

5.2.1. Structure

The most important purpose of the network is to organize the nodes and the sensors. Therefore it always contains the top node and a list of all sensors. The connections between the nodes and those connecting the sensors with the nodes are handled by the nodes and the sensors themselves. The network component only offers an interface to find components by name and to remove or add the top node and/or the sensors. In case

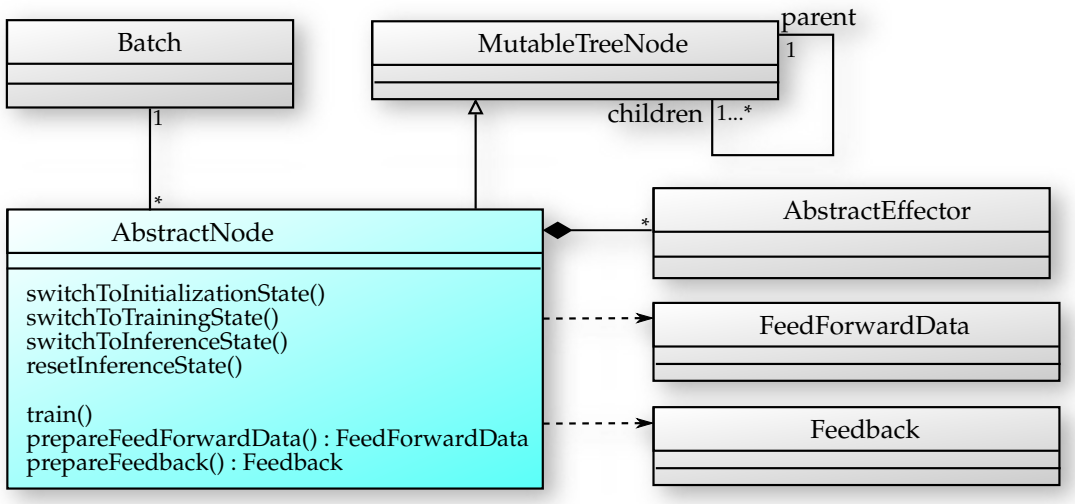


Figure 5.2.: Simplified part of the UML diagram showing the `AbstractNode` class.

of the GUI, the network is enhanced to keep track of all changes amongst its components to offer a common listener interface.

5.2.2. Message Passing

Data is read by the sensors and then passed to the bottom layer of nodes. Those, in return, pass information up to their parents. This process is repeated until the top node is reached. This is the way of the feed-forward data. Some nodes also generate feedback which can be passed down to their children.

Either way, the data is always passed immediately and then stored at the target node. This way, the information is concentrated where it is needed. Thus, the nodes can decide by themselves when they have accumulated enough data to start processing. It will also help to parallelize the software once it is to run on a distributed system. To ensure that no data is being lost, every node has a queue for every child to save feed-forward data and another queue to store feedback messages from its parent.

All the data is conveyed through an instance of either `FeedForwardData` or `Feedback`. They contain an array of double values and a frame counter. The first one encodes the actual data while the second one is just a timestamp. This helps the higher levels to keep track of the current time because they might not be activated every frame.

5.2.3. Nodes

All nodes extend the `AbstractNode` class which provides all the basic functionality for a node in order to work within a network. First of all, it implements the `MutableTreeNode` interface which ensures that the nodes can be arranged as a tree. So every node has a list of children and a maximum of one parent. Furthermore, it provides interfaces to send

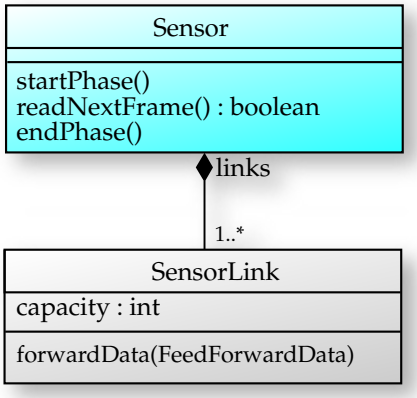


Figure 5.3.: Simplified part of the UML diagram showing the `Sensor` class.

and receive both, feed-forward and feedback data. The `AbstractNode` also takes care of the queuing of the received data. Every time it receives data, it checks if it is able to produce feed-forward data. If it is ready, the current batch is notified and the node gets queued up in the execution queue. In addition to that, the `AbstractNode` manages a list of all attached effectors. It ensures that every effector is handed a copy of every data sent or received by this node.

Every node can be in one of three different states: *Initialization*, *Training* and *Inference*.

Initialization During this state, the node is configured and placed within a network. No processing can be done and it can only transition into training state.

Training Once all the configuration has been done, the node can be switched to Training state. In this state the node is trained.

Inference As soon as the training is completed, the node is switched into this state. From now on, the node is able to perform computations that yield feed-forward data and possibly also feedback.

The transitions between the states are triggered by the batches through one of the following three methods: `switchToInitializationState()`, `switchToTrainingState()`, and `switchToInferenceState()`.

With all this functionality covered by the AbstractNode, the only methods to implement, when writing customized nodes are the `train`(), the `prepareFeedForwardData()`, and the `prepareFeedback()` methods. The methods that switch from one state to another are also commonly overridden, although they need not.

5.2.4. Sensors

Sensors provide the data that is fed into the bottom layer of the network. By now, only file sensors, i.e. sensors that read their data from a file, are implemented. Sensors behave

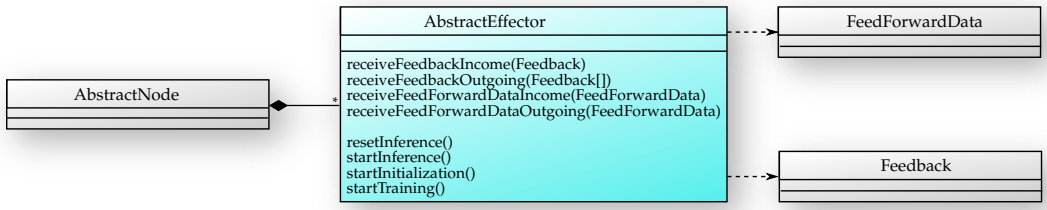


Figure 5.4.: Simplified part of the UML diagram showing the `AbstractEffector` class.

differently from nodes because there is usually only one sensor for several nodes which means that they break the tree structure. Thus, they are also stored as a separate part of the network.

To connect nodes to a sensor, special adapter-nodes called `SensorLink` are used. Every Sensor maintains a list of `SensorLink`s amongst which it distributes the data. Such a link has an associated capacity (i.e. size of the data array to be transferred) and also defines whether the data should be provided as feed-forward or as feedback. From the perspective of regular nodes, the `SensorLink`s are normal children. If a link does not specify a capacity, the sensor automatically determines a suitable value for it.

At the moment, the two foremost important sensors are the `VectorFileSensor` which reads lines of values from a CSV file. And second, the `WindowingSensor` which reads files with only one value per line. It then collects a fixed amount of those values and feeds them as a vector into the network. After that, the first few values of that window are dropped and replaced with the next few.

5.2.5. Effectors

An effector is meant to add some effect (whence the name) to a component. This can be for example printing the data-flow for debugging purposes or measuring the accuracy. In order to do this, every effector receives a copy of all data sent or received by the component it is attached to. Additionally, if it is attached to a node it is notified whenever the node changes its state.

5.3. Batches

Batches are, similar to the network, organized in a tree structure. Every batch can be executed and performs some pre-defined task. After that, it will start executing its child-batches. That means, the batch-tree is executed in a preorder fashion. Some batches collect accuracy information that is upon completion passed to the parent. Those information can be grouped within the tree and result in a matrix of accuracies suitable for plotting.

Every batch can have a list of `Setting`s that can modify the parameters of any component. A `Setting` consists of a regular expression, a parameter name and a value

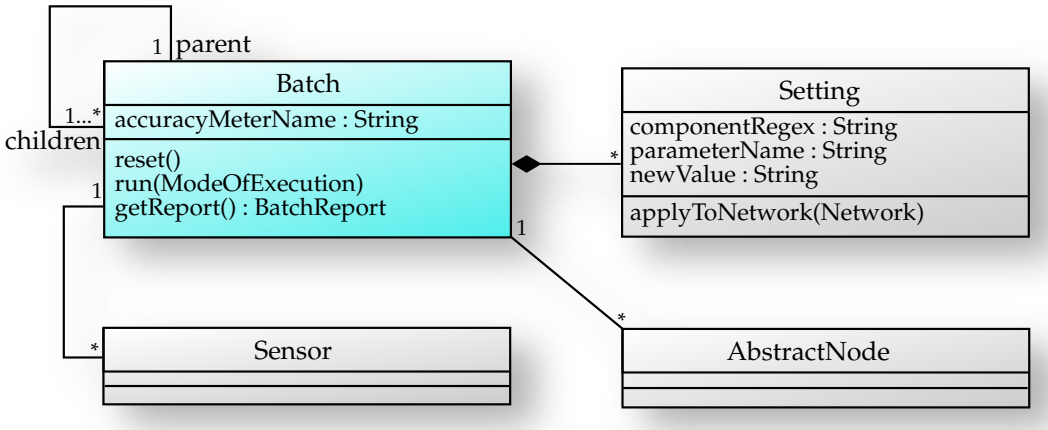


Figure 5.5.: Simplified part of the UML diagram showing the `Batch` and the `Setting` class.

string. By the time the setting is applied to a network, all components whose name match the regular expression will have their parameter with the given parameter name changed to the given value. Currently, the most common settings are those that change the filename parameter of the sensors.

The TestingBatch tests an already trained network. It performs the following steps: First, it applies all the `Setting`s and then initializes the sensors and nodes. Then, for every frame of data read by the sensors, all the nodes perform their computation. Finally, information about the accuracy is collected as far as it is available.

The `TrainingBatch` is used to train any network on a specific data set. To do this, it iterates several training intervals similar to the execution of a `TestBatch`. So it also starts by applying all the `Setting`s and initializing the sensors and the nodes (this time into *Training* state). Then, for every data frame, all the nodes of the lowest untrained layer are processed. After the data file has finished, all those nodes are switched into *Inference* state. As long as there are still untrained nodes, the sensors are re-initialized and the procedure is repeated.

Batches allow stepped execution which means they can interrupt the execution and continue later. This is useful for inspecting and debugging networks and batches using the GUI. The supported modes are:

SingleStep Only perform one step. This corresponds to either executing a single frame of data or the activation of a new batch.

SingleInterval Perform steps as long as the sensors still have data.

SingleBatch Execute a whole batch (for all but `TrainBatch` equal to *SingleInterval*).

Complete Execute the whole batch tree at once.

5.4. HTM Implementation

Before we did any modifications, our first step was to reimplement the NuPIC algorithms. This section explains the algorithms and how they fit into the QHTM framework. Note that we have divided each node in two nodes. One for the spatial pooling and one for the temporal pooling. Pseudocode for all the algorithms can be found in Appendix A on page 69.

5.4.1. Bottom Spatial Pooler

The `BottomSpatialPoolerNode` is equal to the spatial pooler Numenta utilizes for the bottom layer nodes. It has three parameters: *maxDistance*, *maxCoincidences* and *sigma*. Furthermore, it maintains a set of coincidences $\mathcal{C}$. We will now explain all the overridden methods. The rest behaves exactly like the `AbstractNode`.

train

The new input data $\mathbf{x}$ is compared to every stored pattern using the euclidean distance. If it differs from every pattern by more than *maxDistance* and if the total number of patterns is less than *maxCoincidences*, it is added to the list of patterns. Therefore, we get algorithm A.1 on page 69.

prepareFeedForwardData

A probability distribution $\mathbf{y} = \{y_1, y_2, \ldots, y_{|\mathbf{P}|}\}$ is generated by calculating the Gauss function of the distance to the input data for each pattern. Thereby, *sigma* is taken as the standard deviation. After that, the distribution has to be normalized. So we have algorithm A.2 on page 69.

5.4.2. Mid Spatial Pooler

The `MidSpatialPoolerNode` corresponds to the spatial pooler Numenta utilizes for all but the bottom layer nodes. It works similarly to the `BottomSpatialPoolerNode`. It also has three parameters: *maxDistance*, *maxCoincidences* and *poolingType*. The last one can be set to one of the two values, *"Sum"* or *"Product"*.

train

Normally, the feed-forward data of all m children is concatenated to receive the array $\mathbf{x}$. This time, the individual parts $\mathbf{x}^1, \mathbf{x}^2, \ldots \mathbf{x}^m$ are treated separately. They are preprocessed, so that the component with the former biggest value is set to 1 while all the other components are set to 0. Those new arrays are then concatenated as usual. Otherwise, the algorithm is the same as for `BottomSpatialPoolerNode`. Therefore, we just state the preprocessing algorithm A.3 on page 70.

The *maxDistance* parameter has only limited use because of this preprocessing. If only a single part is different after preprocessing that yields a distance of 2. So only multiples of two are reasonable values (if any) for *maxDistance*.

prepareFeedForwardData

During inference the `MidSpatialPoolerNode` calculates a probability distribution over all the patterns. This is done by multiplying each stored pattern with the input data[8]. The components of the resulting vector are, depending on the poolingType parameter, either summed (SUM) or multiplied (PRODUCT). The obtained value constitutes one component in the pattern distribution that is normalized afterwards. Also compare algorithm A.4 on page 70.

5.4.3. Temporal Pooler

On top of every spatial pooler, there is a `TemporalPoolerNode`. This keeps track of the transitions between patterns using a so-called transition matrix $\mathbf{T} = [t_{ij}]_{|\mathbf{P}|\times|\mathbf{P}|}$. Every entry t_{ij} corresponds to the number of transitions from coincidence $\mathbf{c}_i$ to coincidence $\mathbf{c}_j$ counted within the training data. At the end of the training phase, this matrix is used to calculate a similarity measure for clustering the coincidences into so-called *temporal groups* using a *Hierarchical Agglomerative Clustering* algorithm. The number of groups is the only parameter of the node and it is called *maxGroups*.[9]

train

During training the transition matrix is built. All of its entries are initialized to 0. For every input data, we remember the winning pattern, in order to increase the entry of $\mathbf{T}$ corresponding to the transition from the last pattern to this one. The details can be seen in algorithm A.5 on page 70.

switchToInferenceState

Unlike the previous implementations, the `switchToInferenceState` method is overridden by the temporal pooler. That is because at the end of the training, the coincidences need to be clustered into temporal groups. For that, the transition-matrix is normalized such that each column sums up to one. This normalized transition matrix $\hat{\mathbf{T}}$ provides an estimate for the transition probability from one coincidence to another.

What follows is a *Hierarchical Agglomerative Clustering* algorithm that groups the coincidences together into so called temporal groups. It starts with each coincidence in a separate group and then repetetively merges together the two groups that have the highest maximal transition probability to one another. In other words, it uses the maximum of

[8] Recall that the patterns consist only of the values 0 and 1. Therefore, the multiplication yields a similar array but the 1s are replaced by the corresponding values from the input data.

[9] Numenta uses much more complex algorithms with lots of parameters. Given the fact that we replace this node anyways, we decide to stick to the simple algorithms described in George [2008].

the transitional probabilities between the coincidences of this one group and those of the other one as a measure of similarity. The merging is continued until the number of groups matches the *maxGroups* parameter (compare algorithm A.6 on page 71).

prepareFeedForwardData

The feed-forward data of the `TemporalPoolerNode` consists of a vector with one component for each temporal group. Those values are calculated from the coincidence probability distribution received from the `SpatialPoolerNode` as follows: The belief for a temporal group is the maximal probability over all contained coincidences. The overall probability distribution can be produced by normalizing the vector of beliefs (see algorithm A.8 on page 71).

5.4.4. Supervised Mapper

Unlike other nodes, the `SupervisedMapperNode` receives category labels for every frame in addition to the regular feed-forward data. From the regular input data only, the index of the winner is of interest because it is counted how often each winner corresponds to each class label. These numbers are stored in the mapping table $\mathbf{M}$. During inference the class-label most frequently associated to the current pattern is returned.

train

During training, the mapping table $\mathbf{M}$ is filled. This simply means determine the winning pattern and the current classlabel and count up the corresponding entry of $\mathbf{M}$ as shown in algorithm A.9 on page 72.

prepareFeedForwardData

During inference, the index of the winning pattern is used to select the classlabel which most frequently coincided with it. The details are shown in algorithm A.10 on page 72.

5.5. Extensions

The new algorithms we have implemented also fit in the same infrastructure as before. In addition to the methods before, we will present the unfolding algorithms and, obviously, we also have to implement the `prepareFeedback` method.

5.5.1. Spatial Pooler

The `QSpatialPoolerNode` is basically the same as the `SpatialPoolerNode` except for the feedback-related computations. So during inference the only change is to multiply the parental feedback $\boldsymbol{\psi}$ (if any) with the coincidence distribution before passing it up. Otherwise, it is exactly the same as algorithm A.2 on page 69.

prepareFeedback

The `QSpatialPoolerNode` only forwards the feedback $\boldsymbol{\psi}$ received from its parent to its children. This involves translating the distribution over coincidences into distributions over patterns that are understood by the children. This is implemented by averaging all coincidences weighted by the probability induced by $\boldsymbol{\psi}$. The resulting vector has to be split up into m parts, one for each child and each of the parts has to be normalized. The details can be seen in algorithm A.11 on page 72.

unfold

The unfolding method of the `QSpatialPoolerNode` receives the ID of a coincidence to unfold. In addition to that, a time-stamp is provided which is only forwarded to the children. The procedure works as follows. The specified coincidence is split into one piece for each child which is then normalized. Each child is instructed to unfold the winning patttern of its part. Finally, the resulting sets of channels are united and returned. The complete algorithm can be seen in A.13 on page 73.

5.5.2. Sequencer

The most important change in contrast to Numenta's algorithms is the substitution of the `TemporalPoolerNode` with the `QSequencerNode`. This node is placed on top of a QSpatialPoolerNode and performs the temporal inference. It is based on the discrete sequence prediction algorithm *Prediction by Partial Match* (PPM) written by Begleiter et al. [2004].

train

During training, the Prediction by Partial Match (PPM) predictor is trained. Only the winner of each coincidence distribution is used for training because PPM operates on discrete sequences only. These winners are also stored in order to perform the sequence grouping once the predictive entropy can be calculated (see algorithm A.14 on page 73).

switchToInferenceState

Once the training is completed, we can use the PPM predictor to make "predictions" for every coincidence of the training sequence. These distributions can be used to calculate the predicition entropy (also for every symbol/coincidence of the training sequence) which is used to form the groups. For details, see algorithm A.15 on page 74.

prepareFeedForwardData

This method is called whenever the node receives a distribution over coincidences from the spatial pooler below. This input data is then appended to the short-term memory (STM) which also provides the context for the prediction. If the entropy of the prediction

is high, a distribution over sequences is calculated and acts as feed-forward data. So this method does not always generate output (compare algorithm A.16 on page 75 for details).

prepareFeedback

The generation of feedback depends also on the value of the entropy calculated previously by the `prepareFeedForwardData()` method. If the entropy is high and we have some feedback from above, then we use that to calculate our own feedback. Otherwise, the prediction of the PPM about the next coincidence acts as feedback (see algorithm A.17 on page 75).

unfold

This method of the QSequencerNode receives the ID of the sequence to unfold as well as the number of its start frame. The ID is used to determine the coincidences that have to be unfolded by the children. If this node passed any data upwards at any frame, then the division memory would contain the number of that frame. So we can look up the first number in the division memory that is bigger than the given start frame and receive the end frame of the sequence. Therefore, we also know the length of the unfolded sequence. So we can just unfold the coincidences of the given sequence until the result reaches the desired length. Details can be found in algorithm A.18 on page 76.

5.6. Graphical User Interface

We implemented a comprehensive graphical user interface to allow quick configuration and testing of different networks and batches. Its four main components can be seen in figure 5.6 marked with different colors. The “Node View” (green) displays the network as a tree, showing the names of the nodes, their type and their state. It provides functionality to add and remove nodes, sensors, sensor links and filters. Furthermore, networks can be loaded and saved easily. The “Batch View” (cyan) provides similar functionality to edit batches. Here batches can be load, saved, created and deleted. Moreover it provides a possibility to run individual batches or groups of batches using their different modes of execution. That way it is possible to step through the execution and watch exactly what happens within the nodes. Within the “Component View” (red) details about the currently selected component are displayed. It furthermore offers the option to modify the type and the parameters of that component. Finally, all the logging messages can be seen, sorted and filtered using the “Log View” (blue).

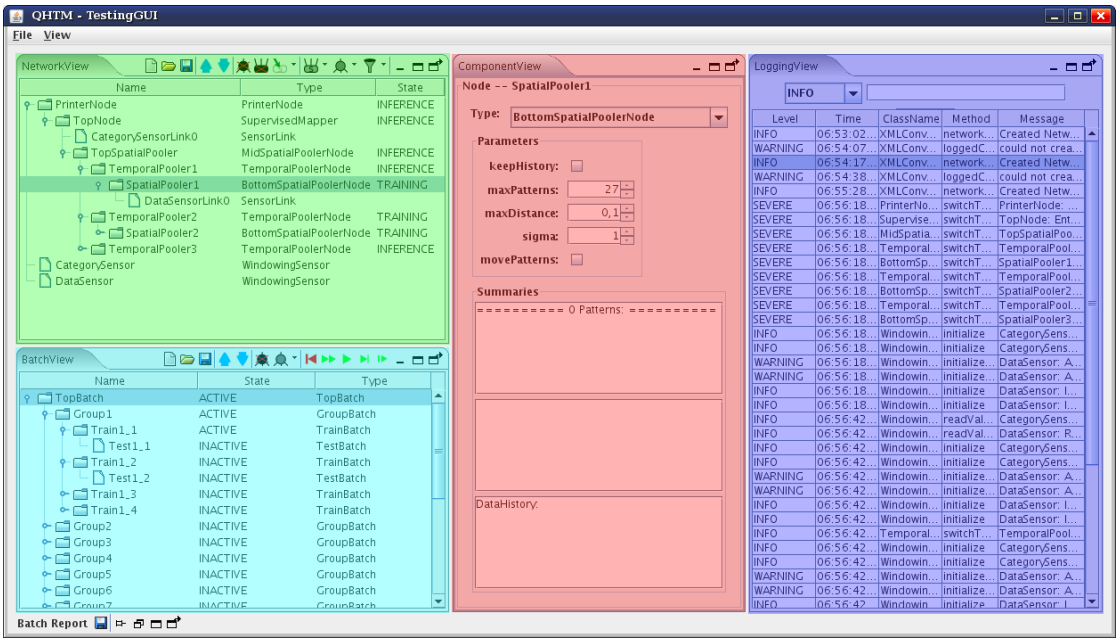


Figure 5.6.: Screenshot of the graphical user interface. The four main components are: Node View (green), Batch View (cyan), Component View (red) and Log View (blue).

6. Results

In this chapter, evaluation results on artificial data for our Algorithms are presented. We start by introducing the hierarchical data sets that we generated for the testing. Then, we show results for unmodified HTMs highlighting the problems they have with this data. We then proceed to show that our algorithms work as expected. We examine their performance in three different ways and study both their advantages and problems. The chapter is concluded by a comparison to the results of conditional random fields and hidden Markov models using our data sets.

6.1. Artificial Hierarchical Data

As testing data, we generated integer sequences that carry various levels of hierarchical structure. The data is designed to be quickly assessable for testing while still covering the main challenges this work is up to.

The data was generated level-wise as follows. First, we use a simple alphabet of integers to assemble a number of sequences which are allowed to differ in length, the so-called *blocks* of the first *level*. The next level is generated using the set of level one blocks as an alphabet by concatenating them and thereby constituting longer sequences. This process is repeated several times until, finally, several instances of the highest-level blocks are concatenated in a random order to form a single, long sequence that builds the data file. Thereby, the generation of each level is governed by four parameters:

1. The *alphabet size* to be used.
2. The *minimum length* of each block.
3. The *maximum length* of each block.
4. The *number of blocks*.

These parameters and the number of levels determine the properties of every data set. Also, they are the same for all levels, thus forcing the *alphabet size* to be equal to the *number of blocks* (because the blocks of each level form the alphabet of the next level).

For each level an additional category-file is generated with the same length as the data file. The following example will illustrate the concept.

6.1.1. The ABC Data Set

This first generated data set will serve as both a test case for our algorithms and an illustration of the process of data generation. It only consists of a single data file, created with the following parameters:

1. An *alphabet size* of 3.
2. A *minimum length* of 3.
3. A *maximum length* of 3.
4. 3 blocks per level.
5. And 4 levels.

For illustrative purposes, we will use different alphabets for every level instead of using only integers as is the case with actual implementation:

$$\Sigma_1 = \{A, B, C\}, \qquad \Sigma_2 = \{1, 2, 3\}, \qquad \Sigma_3 = \{\alpha, \beta, \gamma\} \qquad \text{and } \Sigma_4 = \{\mathfrak{A}, \mathfrak{B}, \mathfrak{C}\}.$$

With these alphabets we can now generate the blocks. Notice that the blocks are all built using the same scheme:

level 1: $\underbrace{ABC}_{1}$, $\underbrace{BBA}_{2}$ and $\underbrace{CAC}_{3}$.

level 2: $\underbrace{123}_{\alpha}$, $\underbrace{221}_{\beta}$ and $\underbrace{313}_{\gamma}$.

level 3: $\underbrace{\alpha\beta\gamma}_{\mathfrak{A}}$, $\underbrace{\beta\beta\alpha}_{\mathfrak{B}}$ and $\underbrace{\gamma\alpha\gamma}_{\mathfrak{C}}$.

level 4: $\underbrace{\mathfrak{A}\mathfrak{B}\mathfrak{C}}_{\mathrm{I}}$, $\underbrace{\mathfrak{B}\mathfrak{B}\mathfrak{A}}_{\mathrm{II}}$ and $\underbrace{\mathfrak{C}\mathfrak{A}\mathfrak{C}}_{\mathrm{III}}$.

For example, we find that the following sequence of 27 symbols corresponds to the level 3 block $\mathfrak{A}$:

$$\underbrace{\underbrace{\underbrace{ABC}_{1}\underbrace{BBA}_{2}\underbrace{CAC}_{3}}_{\alpha}\underbrace{\underbrace{BBA}_{2}\underbrace{BBA}_{2}\underbrace{ABC}_{1}}_{\beta}\underbrace{\underbrace{CAC}_{3}\underbrace{ABC}_{1}\underbrace{CAC}_{3}}_{\gamma}}_{\mathfrak{A}}$$

The corresponding category files are:

data-file:	A	B	C	B	B	A	C	A	C	B	B	A	B	B	A	A	B	C	C	...
level 1:	1	1	1	2	2	2	3	3	3	2	2	2	2	2	2	1	1	1	3	...
level 2:	α	α	α	α	α	α	α	α	α	β	β	β	β	β	β	β	β	β	γ	...
level 3:	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	$\mathfrak{A}$	...
level 4:	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	...

Hence, we have four different classification tasks for this data file.

6.1.2. Random Data Sets

Using the above explained procedure, we generated four different data sets. Each one with a different set of parameters and otherwise completely random. This means that every block was generated drawing symbols randomly from the corresponding alphabet using an equal distribution. Furthermore, the length was also chosen randomly within the given range.

Because no other restrictions apart from the specified parameters apply, the resulting sequences can contain several difficulties with respect to classification:

1. All the blocks might start with a common prefix or end with a common suffix. This would affect the detection of the transitions.
2. Two blocks might be completely equal, thus distinction would be possible only by context.
3. Some symbols might appear much more frequent than others. Therefore, the uncommon ones would be harder to learn because less examples would exist.

To counter these problems, 20 different data files were created for every set of parameters. As we will see, their difficulty w. r. t. the classification task differs heavily yielding a high variance in some of the presented results. In the following, all four data sets are briefly described. They are numbered following the scheme $minLength$, $maxLength$, $alphabetSize$ and $levels$:

Data Set 3334 Using the same parameters as the ABC data set[1], an average accuracy of $33.\overline{3}$ % for this data set could already be achieved by guessing. It is a small and easy data set in order to perform quick tests.

Data Set 3364 The alphabet size is raised to 6 with this data set. All the other parameters stay the same as with the previous one. This means, there is a bigger number of classes to distinguish from each other. So the accuracy with a trivial guessing approach drops to $16.\overline{6}$ %. Also, the final sequence is made twice as long to produce a comparable number of examples per class label.

Data Set 3684 For this data set the maximum length is set to 6, thus allowing blocks of length 3, 4, 5 and 6. Also, the *alphabet size* is further increased to a value of 8. With those parameters, this data set offers a bigger complexity for the classification task. It is designed to test the block separation. Here, pure guessing would yield an average accuracy of only 12.5 %.

Data Set 3984 The maximum block length is even further increased to a value of 9 while the other parameters are the same as with the former data set. The guessing accuracy still amounts to 12.5 % on average but the classification is more difficult because of the big variation within block lengths.

[1] That is: *alphabet size* of 3, *minimum* and *maximum length* of 3, and 4 levels.

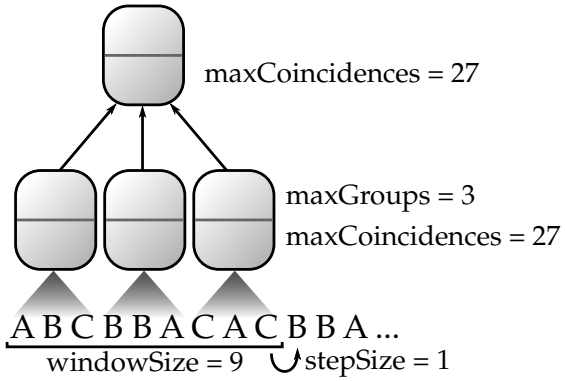


Figure 6.1.: The NuPIC network with two layers used for the experiments. The sequence is shown as a string of As,Bs and Cs at the bottom.

6.1.3. Assumptions

Let us briefly verify that this data set fulfills the assumptions we discussed in subsection 4.1.3 on page 14. Clearly, the ordering of the symbols carries the semantics, so the data is certainly of sequential nature. It is also hierarchically structured in time because we generated it in a hierarchical way. The spatial dimension of the data is very low because we got only one value per time step. But as far as the assumption are concerned, we can say that it has a very primitive spatial hierarchy consisting of only one part. Finally, we have to check whether or not the classification lines up with the hierarchical structure. This is trivially true because we chose the classification to be identifying the hierarchical components (blocks) of a certain level. Hence, all the requirements are met.

6.2. NuPIC Results

We will use the Numenta Platform for Intelligent Computing (NuPIC) to set up a simple network for our data sets in order to prove the problems asserted in section 2.5 on page 7 and to have a base line for our own results later.

The nodes from the NuPIC platform are not directly capable of sequence classification. Therefore, we have to apply some transformation to make it a "regular" classification task. The most simple possibility is to have a fixed-sized window move over the sequence. At every step, the whole content of the window serves as input for the network. We therefore set up a network with two layers as in figure 6.1. The bottom layer consists of three nodes, each one receives three symbols at once, for a total window size of 9 symbols. On top of them, there is a supervised mapper node which additionally receives the category labels. The associated categories are the ones associated with the first symbol of each window.

The network is trained and tested with all 20 data files of each data set. We measure classification accuracy for each of the four category levels, that is the number of correctly guessed labels divided by the total number of labels. The results can be seen in figures 6.2 and 6.3 or as an overview in table 6.1 on page 64. Each of the plots indicates the

average accuracy for every level by a light blue colored bar, whereas the results for the 20 individual files are shown as black dots with gray lines connecting the ones that belong to the same data file.

Looking at the results, we observe that, for all data sets, the accuracy is relatively high for the first level while it drops quickly for higher levels. This is especially true for the more complex data sets. For the higher levels of the third and forth data set, accuracy is very low. This does not come as a surprise because the higher level blocks exceed the size of the window. It thus becomes hard for the network to distinguish them. This could be compensated by increasing the window size. But this leads to other problems. We would have to add more nodes or else the receptive field of every node would grow leading to an exponential increase in possible coincidences. The same holds true for the next layer. If the top-node was to map a large number of nodes onto class labels, it would be very prone to overfitting because a lot more different combinations are possible. So we would have to add more layers even though experiments have shown this to yield bad results. Therefore, this approach is not a good option.

Furthermore, we can see that the variance is highest for the 3334 data set and quite low for the more complex data sets. This is a bit surprising, as for the 3684 and 3984 data sets there is the additional variance of block length within the data. Seemingly, this is more than compensated by the bigger alphabet size. With a bigger alphabet, the blocks become more distinct on average and very similar blocks become unlikely.

It should also be noted that the results are very fragile with respect to parameters. If, for example, the *maxGroups* parameter is increased from 3 to 6, the average accuracies of 82.2 % and 77.9 % for the first two levels of the 3364 data set drop to 67.4 % and 63.0 %.

6.3. Predictions

We start the evaluation of our work by studying the predictions made by the new sequencer node, as they are an integral part of the new algorithms. Our concerns will be if the predictions are valid, how they react to noisy input, if they are able to support the spatial pooler and how they are influenced by the higher levels.

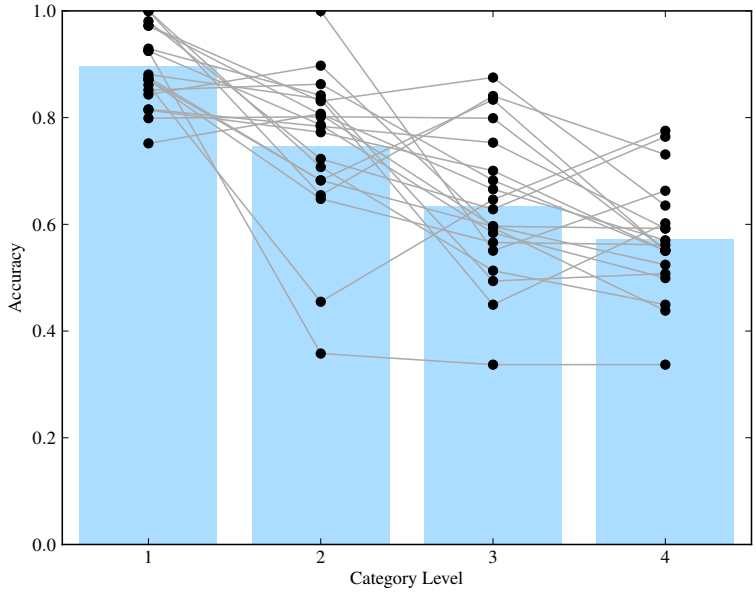
6.3.1. Measuring Predictions

A common measure for the quality of a prediction is the *average log-loss*. It is defined as:

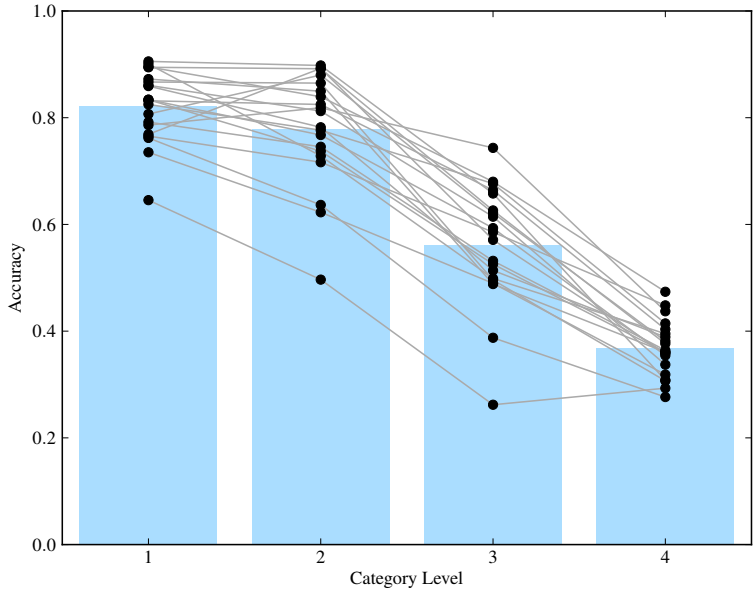
$$\ell(\hat{P}, \mathbf{x}) = -\frac{1}{|\mathbf{x}|} \sum_{i=1}^{|\mathbf{x}|} \log_2 \hat{P}(x_i | x_1 \dots x_{i-1}).$$

Thereby $\hat{P}$ is the conditional probability distribution generated by the predictor and $\mathbf{x}$ is the test sequence. The result of ℓ will be zero if the prediction is always 1 for the right symbol. On the other extreme, it will be infinite if the prediction for any of the symbols is 0.

In the following, we apply the average log-loss to predictions under uncertainty. We use the ABC data set which is encoded as either 0, 1 or 2 (A, B or C) to which we add

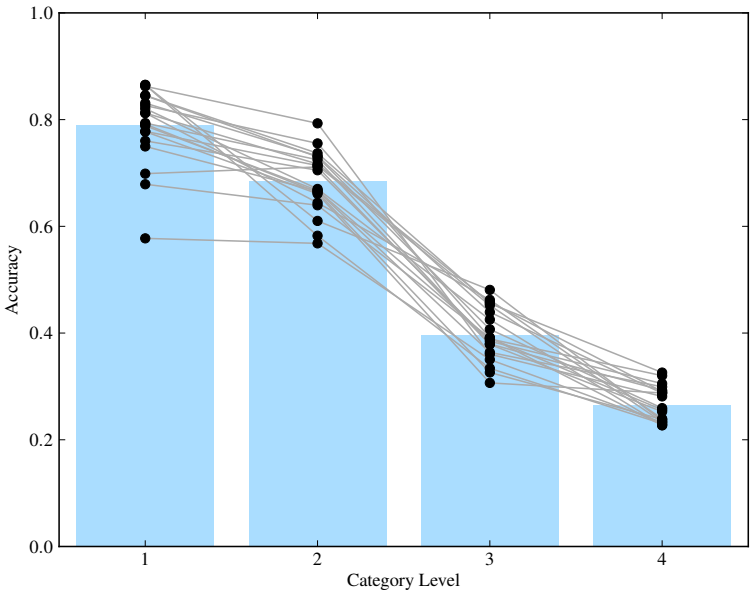


(a) Data set 3334

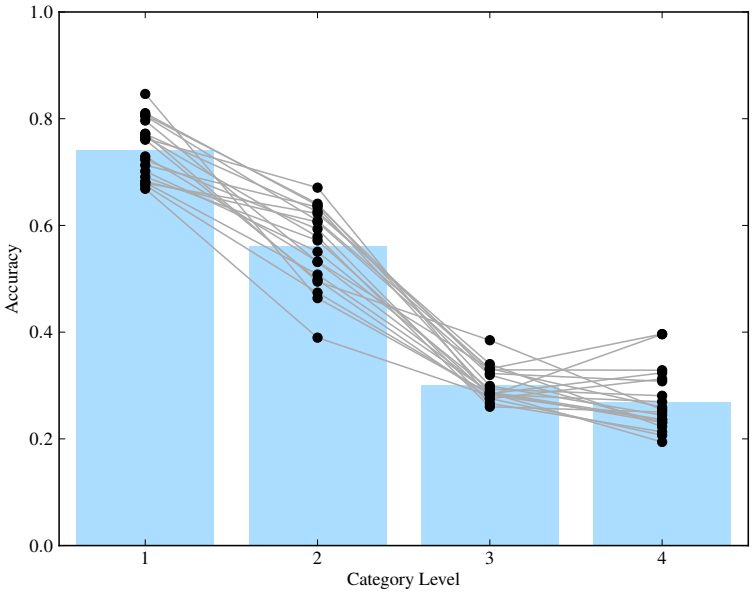


(b) Data set 3364

Figure 6.2.: Results for NuPIC network from figure 6.1 on page 45.



(a) Data set 3684



(b) Data set 3984

Figure 6.3.: More results for NuPIC network from figure 6.1.

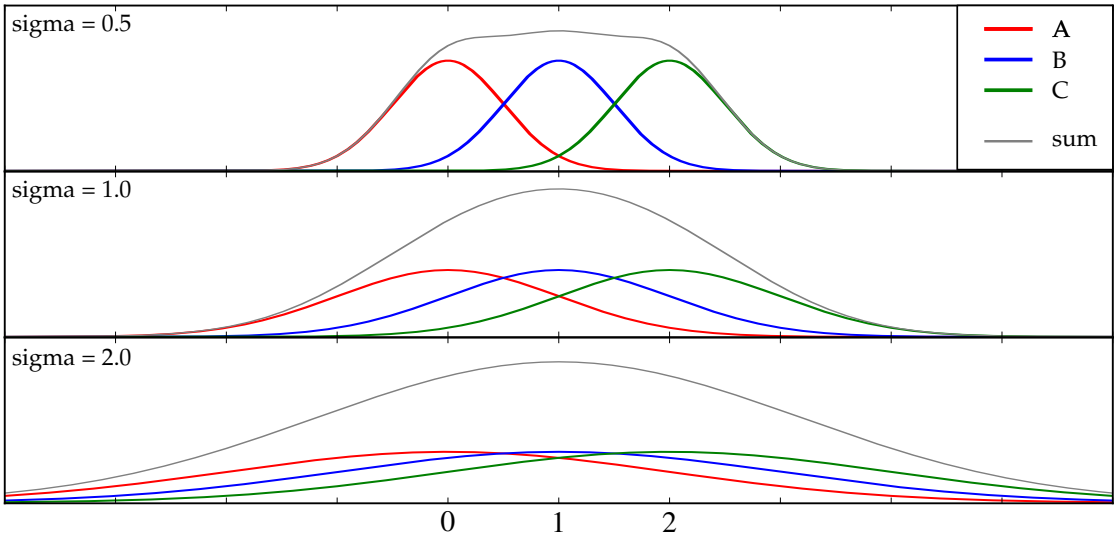


Figure 6.4.: Probability density function for the noisy ABC data.

different amounts of Gaussian noise with a mean of 0 and a standard deviation σ between 0 and 2. The resulting probability density functions are shown in figure 6.4. Note that even with a standard deviation of $\sigma = 1$, one cannot tell apart the three values anymore by looking only at the summed distribution. Reasonable real-data noises will probably be in the area $\sigma \leq 0.5$. So for all the following plots, the interesting part will be the range between 0 and 0.5.

We then take the most straightforward network consisting of only one spatial pooler and one sequencer on top and train it using noiseless data. The trained "network" is then tested on the noisy data. Thereby, the sigma parameter of the spatial pooler is always adjusted to the sigma of the noise. The average log-loss of the equal distribution (which corresponds to pure guessing) is $\log_2 \frac{1}{3} \approx 1.6$. So we want the log-loss of our predictions to stay below this value. The results for Markov-order 1 to 4 predictions can be seen in figure 6.5.

We can conclude that the average log-loss starts quite low and as the noise increases, all of the results approach the value of 1.6 but do not pass it. This tells us that the predictions made by the sequencer are always better than pure guesses. It is also clear that predictions cannot be much better for very high noise because the context the predictions are based on is very uncertain (And making bold guesses about the context does not work out as can be seen in figure 4.6 on page 20.).

Furthermore, we notice that higher orders yield better predictions and that the difference between them vanishes as the noise increases[2]. This is also exactly what we had predicted so, to sum up, this shows that the predictions are working as expected and that they can also cope with uncertainties in a reasonable way.

[2]It is an interesting feature that the line for order 1 crosses those of orders 2 and 3 at about 0.8 and 1.4 respectively. So for very noisy data it is slightly better than the other two. We have no good explanation for this but it could be due to a precision problem.

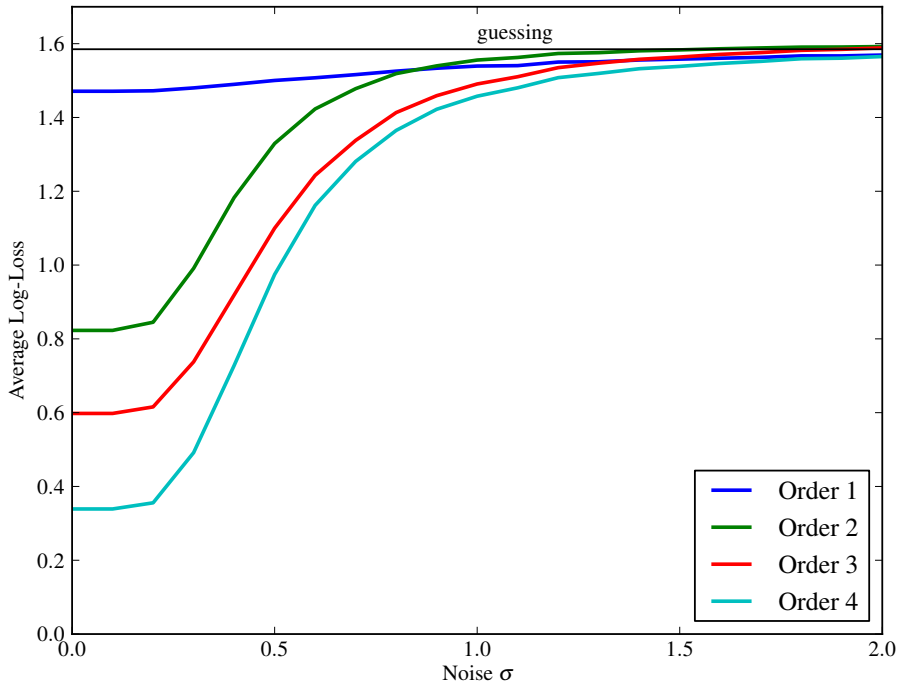


Figure 6.5.: Average log-loss of the prediction utilizing different orders for the VOMM predictor.

6.3.2. Impact of Feedback on Spatial Pooling

Next, we want to verify that the prediction as feedback is able to support the spatial pooler by disambiguating its input. We slightly change the setup, in a way that the log-loss is now determined with respect to the coincidence distribution $\mathbf{y}$ calculated by the spatial pooler. Recall from section 5.5.1 that the calculation of $\mathbf{y}$ includes the feedback. The results can be seen in figure 6.6.

The average log-loss now starts with zero for all curves. This is because in the noiseless case the spatial pooler has no uncertainty whatsoever about the current symbol. As the noise increases, the curves spread out. The average log-loss is then highest for the network without any prediction and lowest for the order of 4. Again, the curves approach (even though much slower) the guessing line but do not cross it. This observation confirms our hypothesis that feedback can significantly improve the performance of the spatial classifier. For the region around $\sigma = 0.5$, the improvement is more than a third of the average log-loss. If the *alphabet size* was much higher, that effect would probably be even more significant.

6.3.3. Multilayer Predictions

Finally, we add further layers to our network to study their impact on the predictions. The resulting "hierarchy" will be a single straight line because we only have one input

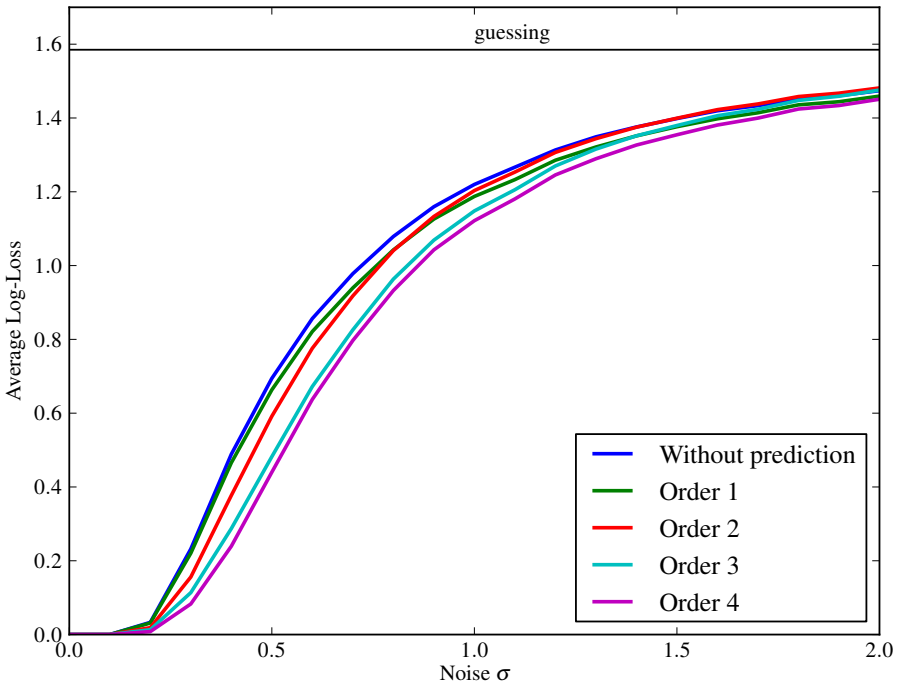


Figure 6.6.: Average log-loss of the coincidence-distribution calculated by the spatial pooler with different orders of supporting predictions and also without.

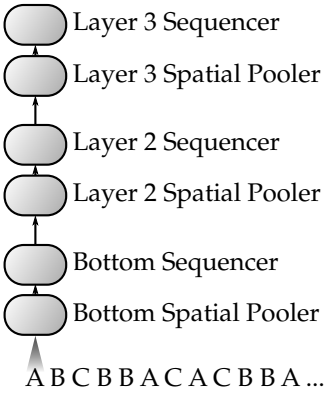


Figure 6.7.: Three-layered network used for measuring the effect of multiple layers upon prediction.

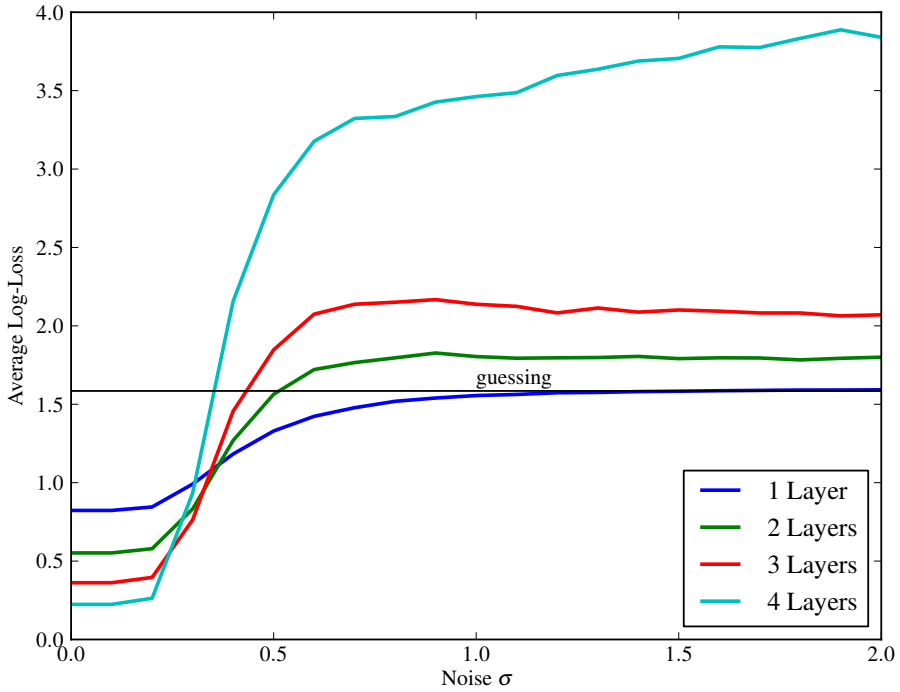


Figure 6.8.: Effect of multiple layers of predictors on the average log-loss of the prediction. The order of the predictors is $k = 2$ for every layer.

value (compare figure 6.7). We expect to see a slight improvement for every layer that is added.

Figure 6.8 shows the results for 1 to 4 layers each with a Markov-order of $k = 2$. In the low-noise area we can see that additional layers improve performance as expected. Also note that the advance is quite significant: If we compare this plot to figure 6.5, we notice that two layers with an order of 2 each already excel one layer with order 3. Likewise, three layers yield better predictions than one layer with an order of 4. However, as the noise increases, the gain quickly turns into a loss. Between $\sigma = 0.3$ and $\sigma = 0.4$ the results turn upside down completely. From thereon, the four-layered network is by far the worst while the single-layered network is the only one not to pass the line of guessing.

To understand the reason for this inversion, we have to consider the implication the noise has on the feed-forward data. The problem is that the noise not only disrupts the context for the prediction but also leads to disturbances with the timing of the feed-forward data. This means that the sequencer will send data upwards more often because the prediction entropy will pass the threshold more frequently. The layer on top then receives data with temporal noise in addition to the spatial noise from the input data. This result is clearly a sign that our algorithms are not robust enough against temporal noise.

The problem is less significant if training is done on the noisy data. Then, the higher levels rely less on the structure and generally make weaker predictions. Figure 6.9

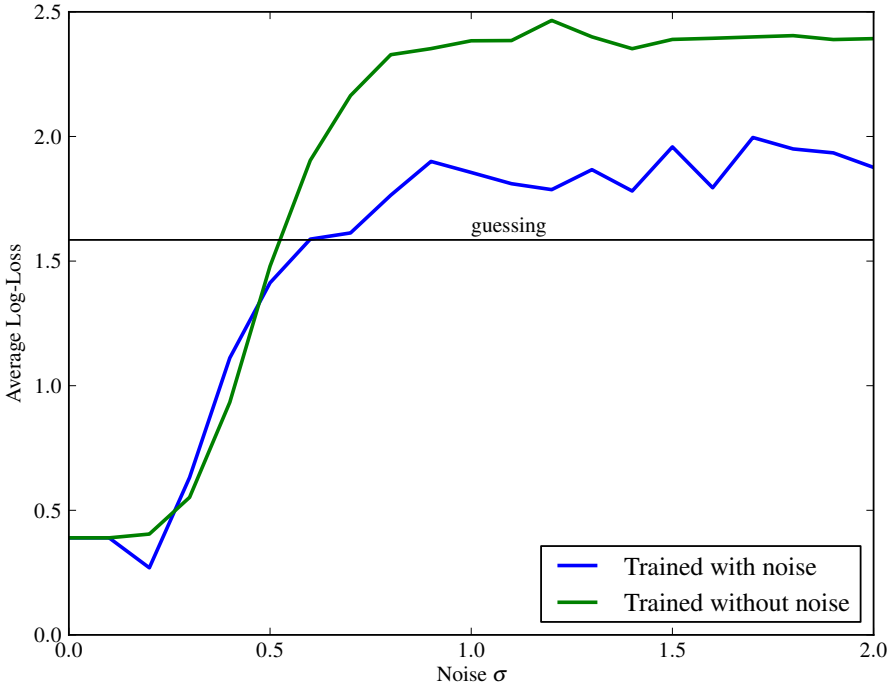


Figure 6.9.: Comparison of training with clean and with noisy data for a three-layered network with $k = 3$.

compares the results of training on clean data with those of training on the noisy data for a network with three layers and $k = 3$. Clearly, the value for $\sigma = 0.2$ is an outlier, while the results starting with $\sigma = 0.5$ show significant improvement.

6.4. Top-Node Classification

In the following, we will study the actual classification capabilities of our work. To this end, we build a network with a *supervised mapper* node on top (compare left side of figure 6.10). This component is from the Numenta framework and is also the way NuPIC networks perform classification. In every frame, it receives a category label in addition to the regular feed-forward data. Its task is to find a good mapping from the feed-forward data to the corresponding class label. This task can be performed by counting how many times each winning pattern of the feed-forward data coincides with each class label. During inference, the class label associated to the currently seen pattern most frequently is chosen and compared to the actual class label to compute the accuracy. The supervised mapper component is explained in detail in section 5.4.4 on page 38.

Unfortunately, this does not work for our algorithms because the top node usually does not receive feed-forward data every time frame (compare figure 6.10). So the relation between input data and class labels cannot be a one-to-one mapping. To solve this problem, we extend the supervised mapper, so that it caches all the intermediate class

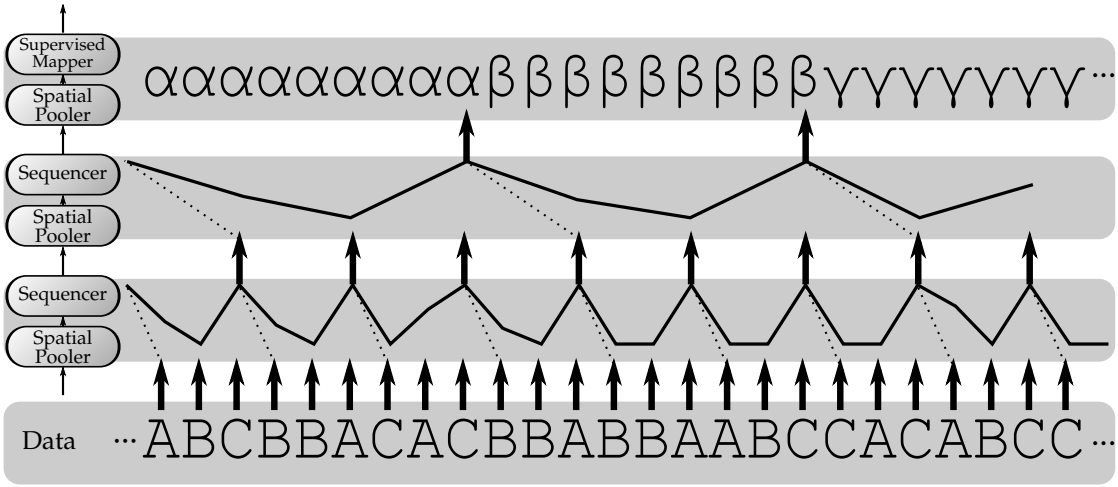


Figure 6.10.: Conceptual visualization of classification using a top node. The network is shown on the left. The rest represents the entropy over time as well as feed-forward timings.

labels. Every time it receives feed-forward data, it is mapped to all of the cached categories. With this configuration, we can now test our algorithms on the generated data sets.

Some results for the 3364 data set are shown in figure 6.11, the rest is summarized in table 6.1. The networks used have one, two and three layers and a supervised mapper layer on top of that. All the predictors use an order of $k = 3$. We can see that each result is reasonably good for only one particular level while it is worse for all the others. This correlation is due to the feed-forward data timing of the network and the supervised mapper which can only map data to blocks of class labels whose size depends on the timing of the feed-forward data. To produce a good mapping, the induced blocks of category data have to coincide with the actual blocks. In other words, the level of abstraction has to match. If the top node receives data that corresponds to level 3 blocks, then they will neither be mappable to level 2 nor to level 4 blocks, at least not without a loss of accuracy. This is exactly what can be seen in figure 6.11. In order to get rid of this limitation, we will take another approach which is classification through reconstruction.

But before, we want to discuss the implications of these results. Consider the fact that the number of layers correlate to the level the network is particularly good at. This shows that at the higher levels more abstract concepts are formed. We can tell that because we know, that the supervised mapper will only achieve good accuracies if the size of the category blocks matches the feed-forward timing. So, this particular accuracy distribution tells us what level of abstraction the formed concepts match best. And this level, as we can see, increases with the number of layers. This is the behavior we were looking for because it clearly shows that the temporal hierarchy works in the desired way.

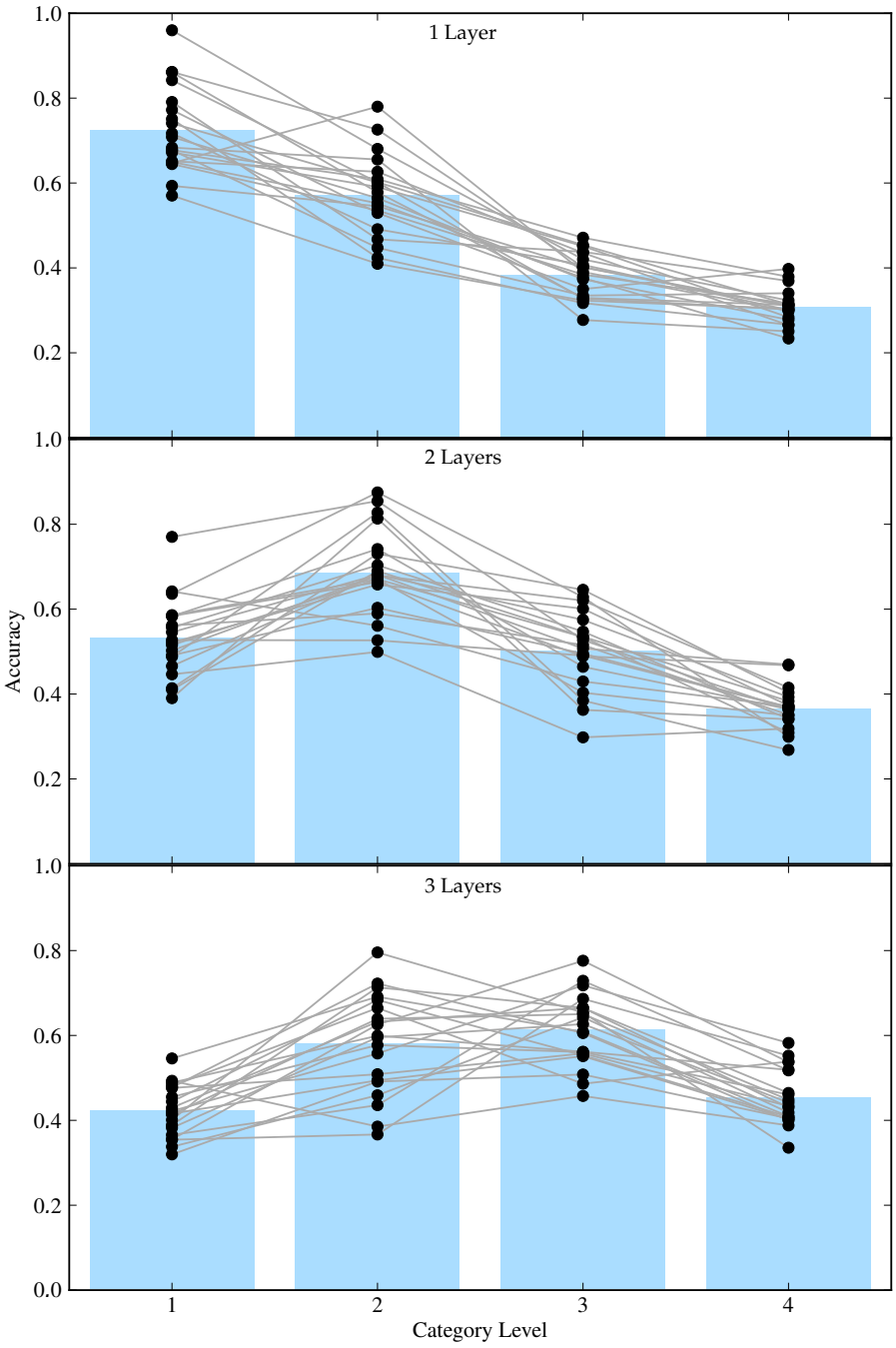


Figure 6.11.: Classification accuracies for the 3364 data set using a network with one, two, and three layers and a supervised mapper on top. All orders k equal to 3.

6.5. Classification through Reconstruction

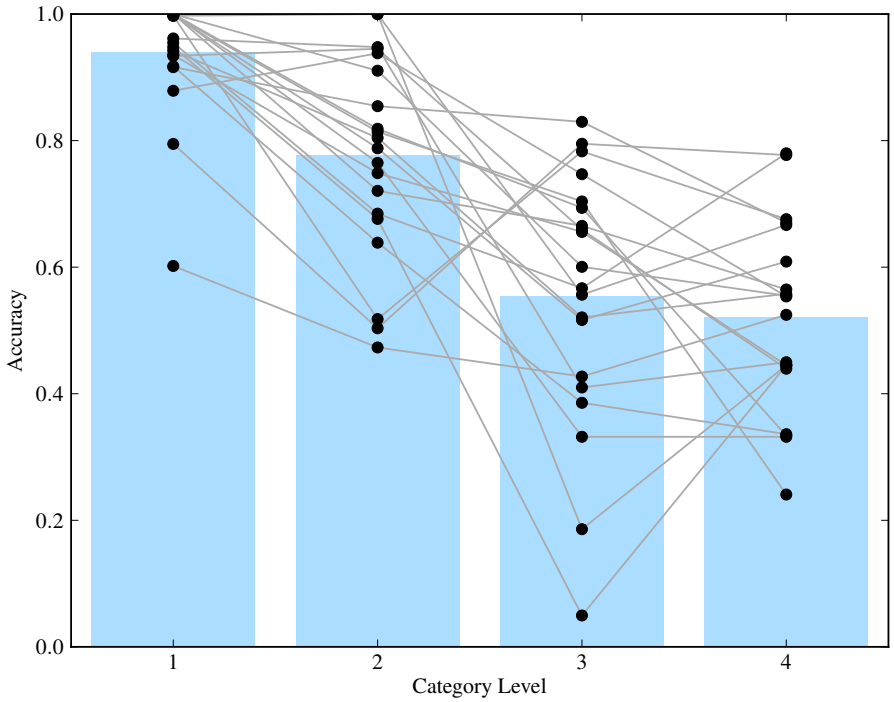
As we have seen, the main problem with a top-node-based classification is that the grouping done by the network has to match the actual blocks of class labels very well. Otherwise, the top-node will not be able to find a good mapping. But consider a network that creates a higher level of abstraction than the class labels require. Using speech recognition as an illustration once more, this would correspond to a network that recognizes whole words, while the task is to classify syllables. In this case, the top-node will struggle, because most words consist of two or more syllables and, therefore, a one-to-one mapping of the words to the syllables is impossible. But obviously, the required information has already been recognized by the network (given the word was correctly classified). The only problem about it is that the syllables are considered details (because they are sufficiently predictable) and are therefore not passed upwards, so they do not reach the top node. By extracting that "hidden" information, that is possibly spread throughout the whole network, we could solve the problem and achieve much better results.

This is where the concept of unfolding comes in (see section 4.7 on page 25). The idea is to combine the input data and the class labels and feed both into the bottom layer at training time. In doing so the network is allowed to build a joint model of class labels and input data. To perform classification, we then leave the class labels away. The network will select the interpretation that it considers to match best, basically treating the missing labels as a form of noise. Thereby, it implicitly fills in the missing category information. So all we have to do is to unfold the high level concepts, the top node of the network emits, back into the corresponding stream of input data. This will then contain the desired category information.

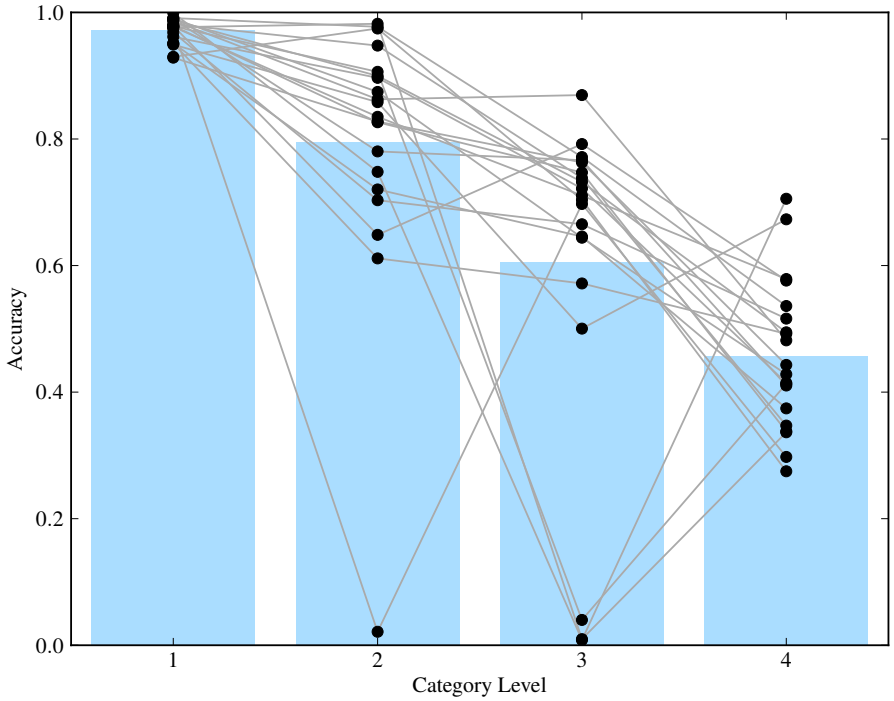
Some of the results for this approach can be seen in figures 6.12 and 6.13. Despite the fact that they stem from one-layered networks, the results are almost perfect for the low levels. The accuracy for the first level is around 95 %, even for the complex data sets. Also for the lower levels, they outperform all of the previous results by far. This shows the impact of the mapping problem.

However, an interesting fact to mention is that this approach is obviously more prone to complete failures. For example, in figure 6.12b) a few results have an accuracy of about 0 % although the average is quite good. Such outliers occur if for some reason the network chooses the wrong interpretation of the "noisy" data. The beginning was probably highly ambiguous and the right interpretation was slightly less likely than some other. In this case, the network will, through feedback, strengthen its own model and stick to it forever. Hence the reconstructed class labels can all be wrong due to one fatal error in the beginning.

If we start adding additional layers to our reconstruction network, another effect cuts in. If we compare the results for two layers with those for only one layer (see table 6.2 on page 65), we can see a significant drop in accuracy for most cases. The reason for this is the same as with the multi-layer predictions (compare section 6.3.3 on page 50). As soon as the noise exceeds a certain value, the negative effects increase each other. The purely spatial noise also triggers a temporal noise due to our feed-forward strategy and soon the predictions are completely useless. The same effect cuts in here because from

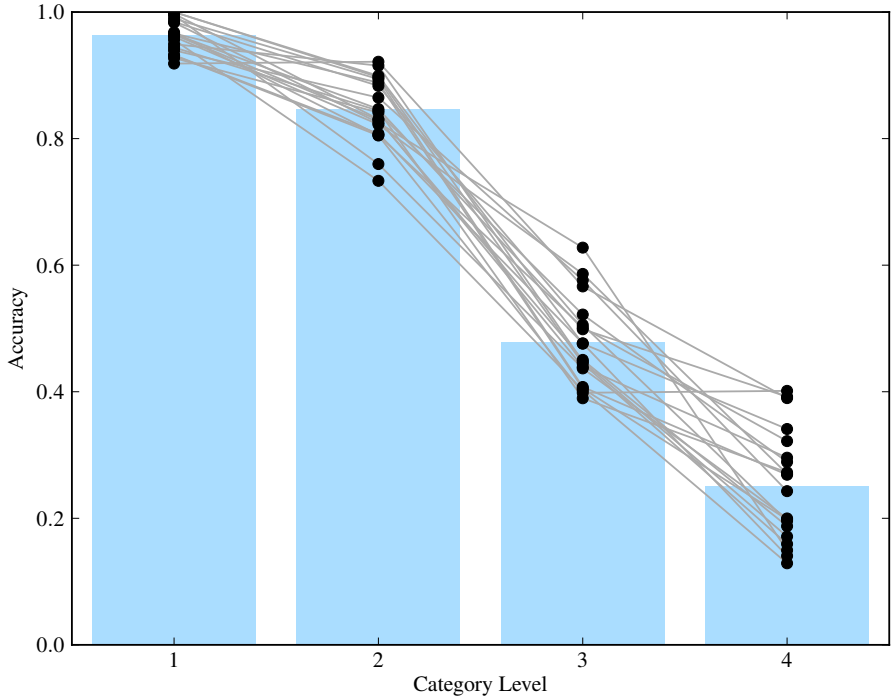


(a) Data set 3334

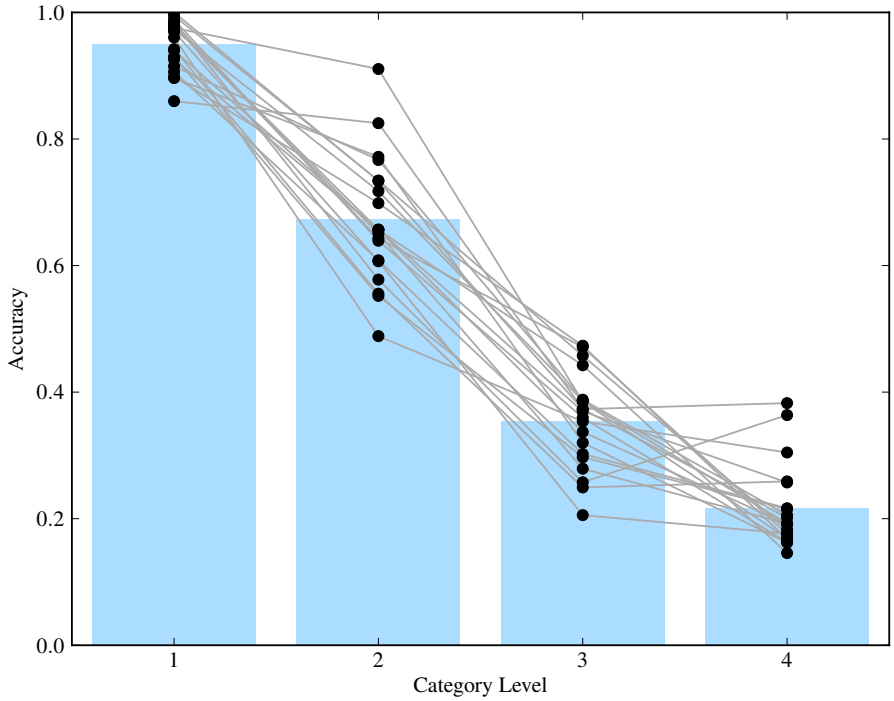


(b) Data set 3364

Figure 6.12.: Results for the unfolding approach. In both cases a one-layer network with a Markov-order of 3 is used.



(a) Data set 3684



(b) Data set 3984

Figure 6.13.: Results for the unfolding approach. In both cases a one-layer network with a Markov-order of 2 is used.

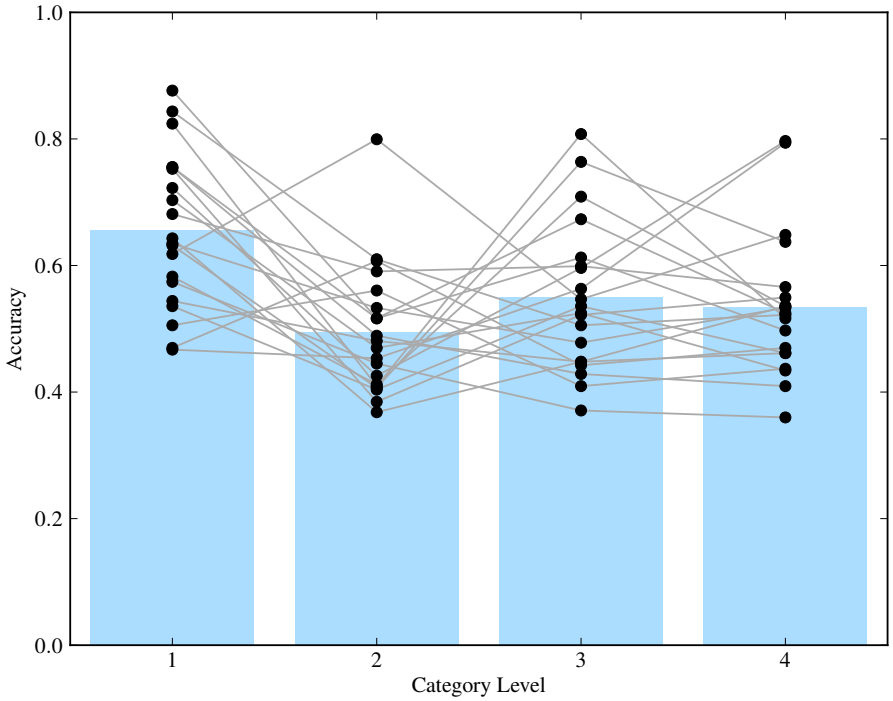
the perspective of the network, missing class labels are nothing but noise. Unfortunately, they correspond to quite a big amount of noise, so that the results are useless.

So we have seen that for the lower levels this approach provides nearly perfect results. For the higher levels, the results are still quite good compared to the other results. But, due to the temporal noise problem, we were not able to build an effective multi-layer reconstruction network.

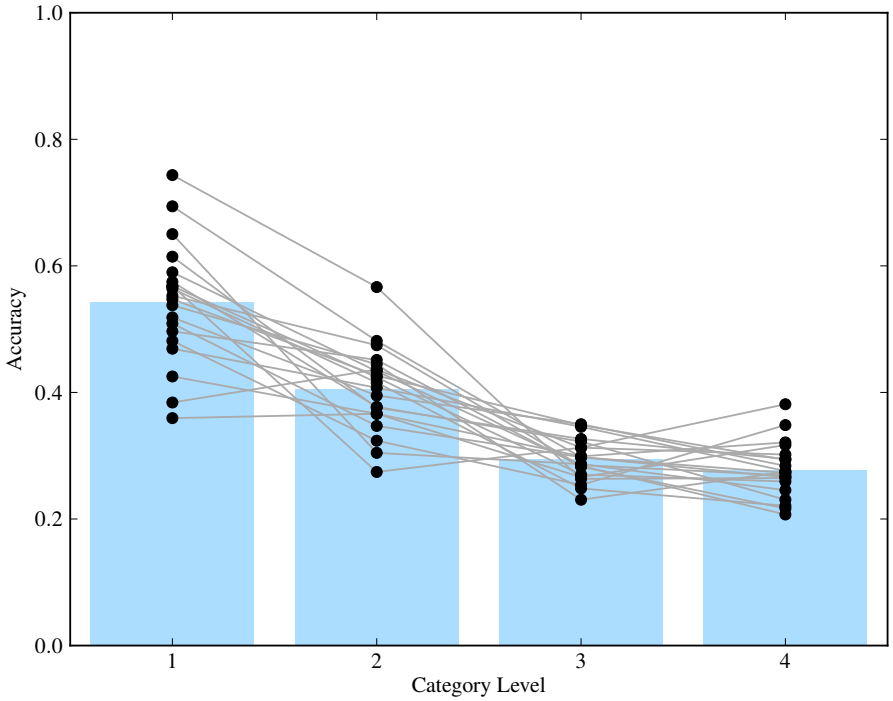
6.6. Comparison

In this last section, we want to present some results for conditional random fields (CRF) and hidden Markov models (HMM), two well-known methods for sequence learning. For this, we used the MAchine Learning for LanguagE Toolkit (MALLET) which supports both CRFs and HMMs. For the tests, we converted our data sets into the required format and ran the software with its default settings. The results can be seen in figures 6.14, 6.15, 6.16 and 6.17. You can also see an overview in table 6.3.

Surprisingly, their results are quite bad. Even for the easiest cases, they only just touch the 66 % accuracy. Most of the time, the results stay between 30 % and 50 %. This might be a result of the fact that both methods only consider first order dependencies. However, it clearly shows that the data sets we generated are not easy to solve.

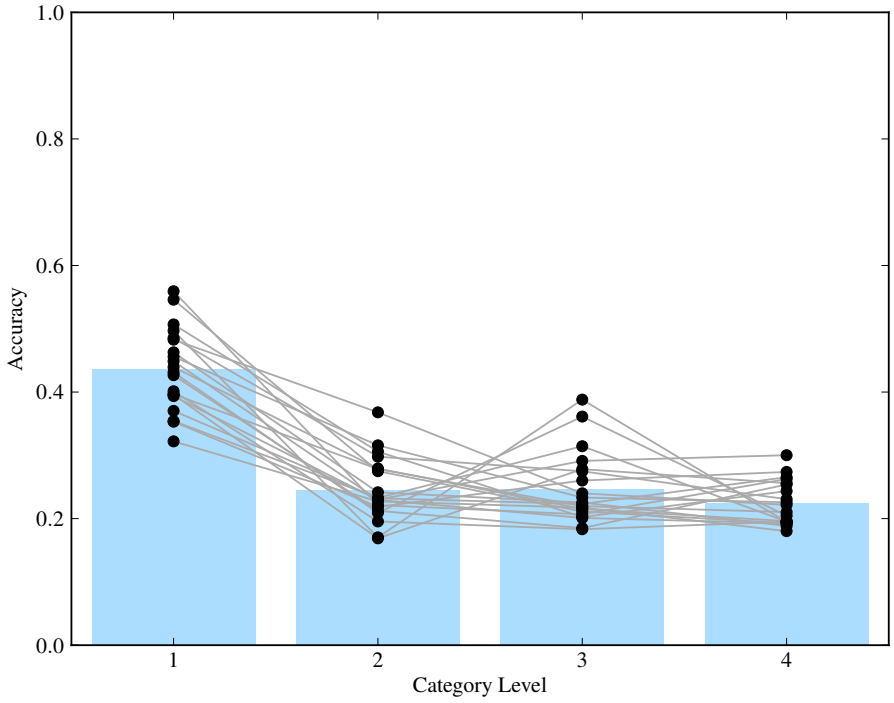


(a) Data set 3334

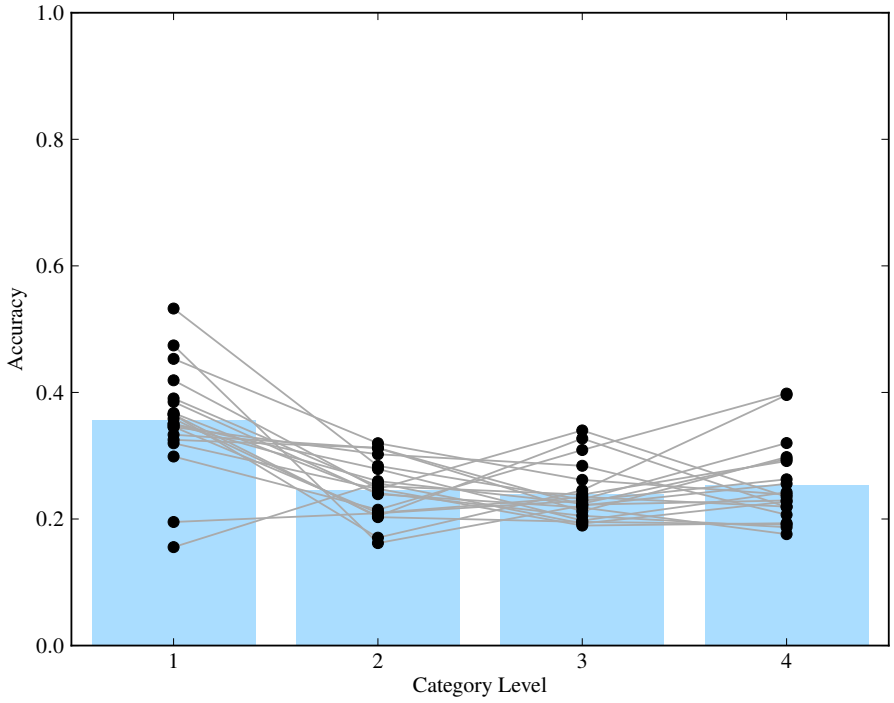


(b) Data set 3364

Figure 6.14.: Results for our data sets using Conditional Random Fields.

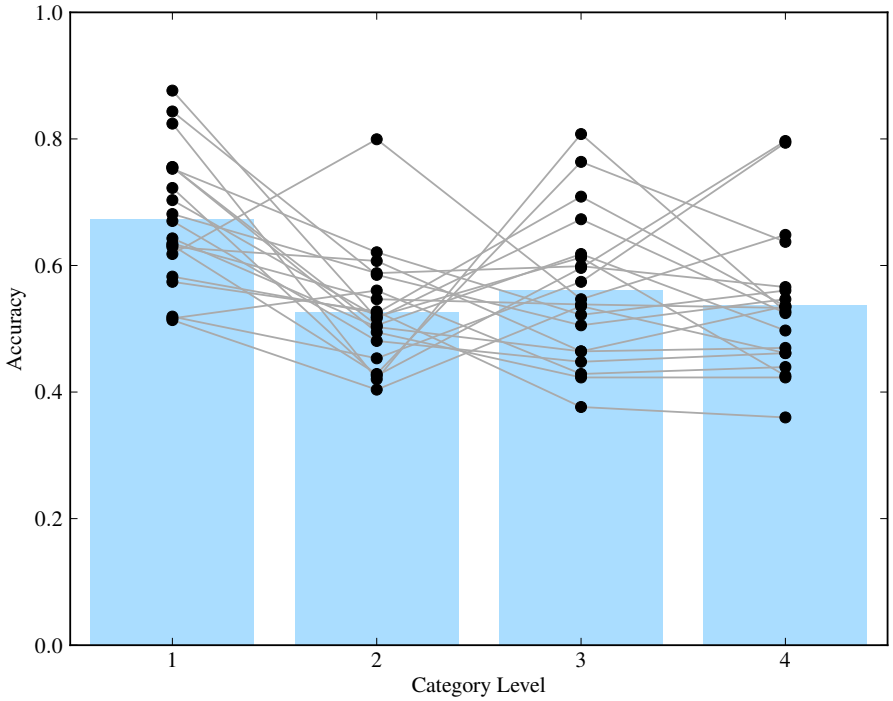


(a) Data set 3684

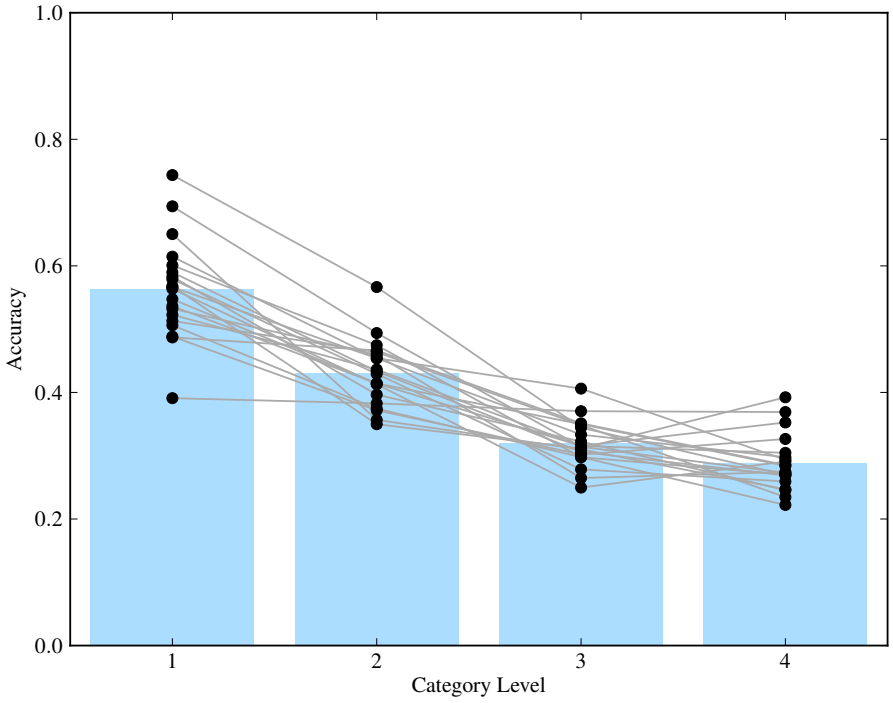


(b) Data set 3984

Figure 6.15.: Results for our data sets using Conditional Random Fields continued.

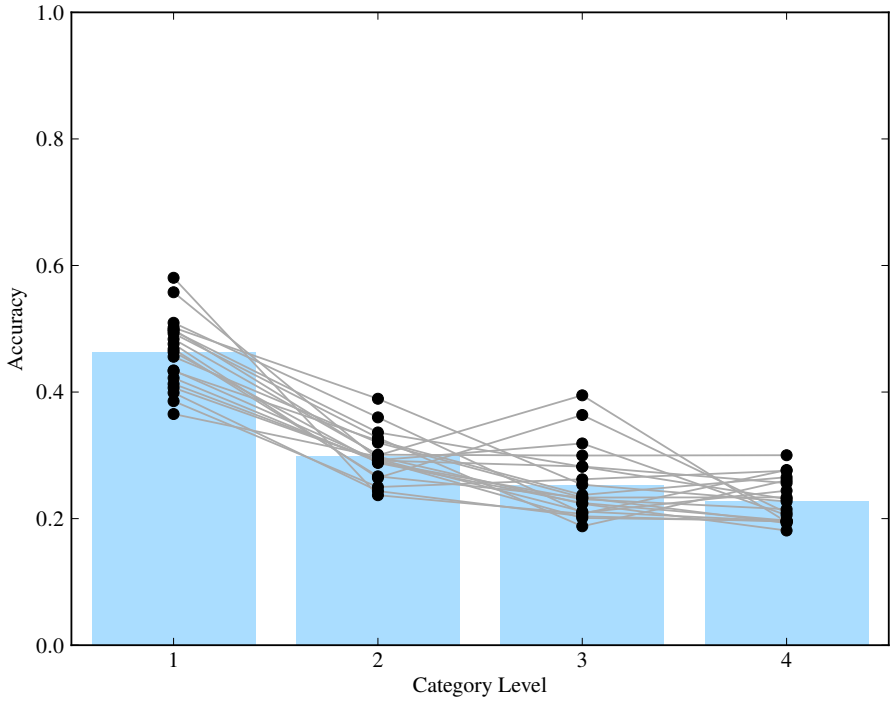


(a) Data set 3334

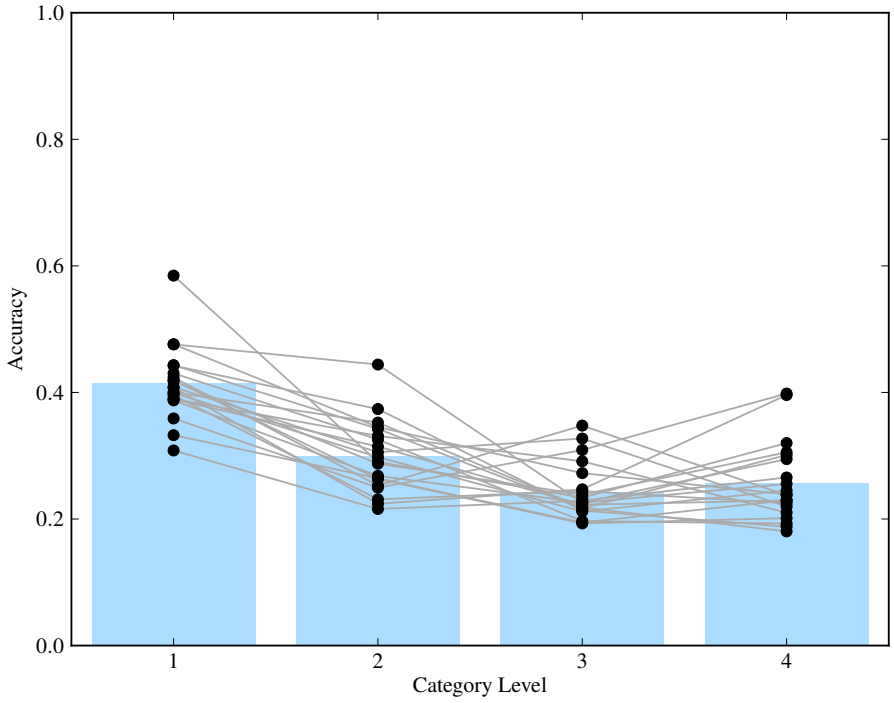


(b) Data set 3364

Figure 6.16.: Results for our data sets using Hidden Markov Models.



(a) Data set 3684



(b) Data set 3984

Figure 6.17.: Results for our data sets using Hidden Markov Models continued.

		Accuracy in %			
Data set	Level	NuPIC	1 Layer	2 Layers	3 Layers
3334	1	**89,7**	72,8	63,2	61,3
	2	**74,7**	62,8	63,8	63,5
	3	63,5	56,8	60,8	**64,4**
	4	57,2	55,2	54,9	**57,9**
3364	1	**82,2**	72,2	53,0	42,1
	2	**77,9**	56,9	68,4	58,0
	3	56,1	38,1	50,1	**61,1**
	4	36,8	30,6	36,4	**45,3**
3684	1	78,9	**84,8**	54,4	41,2
	2	68,5	45,7	**71,0**	66,1
	3	39,7	30,2	41,6	**53,0**
	4	26,6	24,1	27,4	**31,7**
3984	1	69,2	82,4	61,6	42,7
	2	**66,5**	43,0	61,7	65,7
	3	40,1	27,2	32,5	**44,7**
	4	**29,6**	25,8	26,6	28,4

Table 6.1.: Overview comparing the NuPIC Results to those created by a network with one, two and three layers and a supervised mapper on top. All orders k for that network equal to 3.

		Accuracy in %								
Data set	Level	NuPIC	O1	O11	O111	O2	O22	O222	O3	O33
3334	1	89,7	81,8	66,5	66,5	90,3	87,6	60,5	**93,9**	91,7
	2	74,7	70,2	48,6	48,6	73,0	73,5	46,1	**77,7**	71,9
	3	**63,5**	55,9	34,7	34,7	51,6	41,0	41,3	55,4	49,4
	4	**57,2**	57,3	43,7	43,7	45,7	39,2	33,1	52,0	34,3
3364	1	82,2	82,2	36,2	36,2	94,1	90,1	25,6	**97,2**	93,3
	2	77,9	74,4	37,4	37,4	78,5	72,0	27,8	**79,5**	72,7
	3	56,1	49,8	32,6	32,6	60,2	46,4	22,8	**60,5**	44,3
	4	36,8	42,3	24,6	24,6	44,9	36,4	25,1	**45,7**	29,1
3684	1	78,9	87,7	74,9	32,1	**96,4**	*	*	*	*
	2	68,5	79,6	60,4	28,0	**84,6**	*	*	*	*
	3	39,7	**49,1**	33,2	23,1	47,8	*	*	*	*
	4	26,6	**31,2**	22,1	27,5	25,1	*	*	*	*
3984	1	69,2	88,6	79,1	32,2	**95,0**	*	*	*	*
	2	66,5	**69,7**	42,6	25,5	67,2	*	*	*	*
	3	40,1	**40,3**	29,9	20,9	35,4	*	*	*	*
	4	**29,6**	28,4	24,1	24,2	21,6	*	*	*	*

Table 6.2.: Overview comparing the NuPIC Results to those created by a network with one, two and three layers unfolding the class-labels. The stars indicate tests that could not be finished in time.

		Accuracy in %				
Data set	Level	NuPIC	CRF	HMM	SM O333	Unfolding O2
3334	1	89,7	65,6	67,2	61,3	**90,3**
	2	**74,7**	49,4	52,6	63,5	73,0
	3	63,5	54,9	56,0	**64,4**	51,6
	4	57,2	53,4	53,7	**57,9**	45,7
3364	1	82,2	54,2	56,4	42,1	**94,1**
	2	77,9	40,5	43,0	58,0	**78,5**
	3	56,1	29,4	31,9	**61,1**	60,2
	4	36,8	27,7	28,8	**45,3**	44,9
3684	1	78,9	43,6	46,2	41,2	**96,4**
	2	68,5	24,5	29,8	66,1	**84,6**
	3	39,7	24,7	25,3	**53,0**	47,8
	4	26,6	22,4	22,8	**31,7**	25,1
3984	1	69,2	35,6	41,5	42,7	**95,0**
	2	66,5	24,6	29,9	65,7	**67,2**
	3	40,1	23,9	24,3	**44,7**	35,4
	4	**29,6**	25,4	25,7	28,4	21,6

Table 6.3.: Comparison of one representative per method.

7. Conclusion and Perspective

This thesis has shown how to extend the HTM technology in order to improve its sequence classification capabilities. This has been achieved. We have created the Sequencer, a component that replaces Numenta's temporal pooler. Unlike the latter it is able to do sequence learning and to make predictions about the next pattern. We integrated it into the HTM framework and added a feedback mechanism to enhance the collaboration with the spatial pooler. Furthermore, we implemented pattern unfolding, a technique for data reconstruction and noise removal. Finally, we evaluated our work using artificial data sets and compared the results to those of unmodified HTMs using a window, hidden Markov models and conditional random fields.

We were able to show that our concepts work as expected. We have demonstrated the positive effect feedback in the form of predictions has on the recognition of spatial patterns. Moreover, we showed that a deeper hierarchy can further increase that positive effect, as long as the noise stays below a certain threshold. We utilized two different approaches to sequence classification: top-node based classification and classification through reconstruction. Both were able to outperform conditional random fields and hidden Markov models, and, in many cases, also showed a significant increase over unmodified HTMs utilizing a small window. The evaluation also revealed some weaknesses of our algorithms. In particular the problems of the sequencer in dealing with temporal noise are quite profound as they also limit the performance of classification by reconstruction.

In addition to the empirical results, we believe that this thesis has provided a general method to strengthen the cooperation of spatial and temporal learning algorithms. The topic of this thesis was chosen to have a good starting point for the exploration of this complex issue and the main achievements of this work are the abstract concepts and implications of our results. We want to specifically point them out here:

First of all, we have shown a way to use a feedback mechanism to have temporal inference support spatial inference, the operability of which does not depend on specific methods. Any unsupervised classification algorithm that is able to output a distribution over the learned classes can be used. Likewise any sequence learning algorithm that is able to make predictions about the next symbol is suitable. This means that this kind of cooperation is applicable to a wide range of problems.

Furthermore, we provided some insights about the process of modeling a temporal hierarchy which is complementary to the spatial hierarchy already modeled by HTMs. In our opinion, the combination of those two is a key concept. Because, if we were able to split our problem hierarchically, both in space and time, then we can apply our pair of methods to the smaller sub-problems. That way the cooperation takes place at this much smaller scale, which will presumably increase the positive effects.

We found that it was necessary, in order to build a temporal hierarchy, to adjust the partitioning of the sequence to the actual data. Otherwise we would not be able to create an efficient model. Moreover we presented a way to find reasonable partitions using only the predictions of the sequence learner. Notice that again the concept is not dependent on any specific sequence learning algorithm. In our case we used an implementation of the prediction by partial match algorithm, but we could just as well have used any other sequence learning algorithm. As long as it can be trained unsupervised and it provides reasonable predictions based on a (small) context, the method applies.

We also discovered an important problems related to this approach: It is fragile to temporal noise. Unfortunately we did not have the time to study this in detail. Therefore it will be left for future work to figure out ways to fix this problem.

So in summary, we were able to take the first steps towards a framework which closely combines spatial and temporal inference. But obviously this goal is far to ambitious for a thesis like this. Therefore it is clear that many things remain, we did not cover. For example, we did not have a real spatial hierarchy because our sequences consisted of single integers only. Thus, we could not examine the interplay between a spatial and a temporal hierarchy. This would probably be both very interesting and also quite difficult to study. It would require a very careful setup and thorough testing, to pin-point occurring problems. Also synchronization of different nodes would become an issue. In our one line hierarchy every node had exactly one child. Therefore the case that some of the children sent feed-forward data, while others did not, could never occur. But if we ran a network with a "real" hierarchical structure, we would have to deal with this situation.

Another interesting issue would be to also apply the cooperation of spatial and temporal methods at training time. If the noise-reduction effect we observed during inference transfers to training time, the internal models of both methods could be built upon much cleaner data. Possibly, this would also lead to a facility for online-learning. Future investigation should, furthermore, focus on optimizing the run-time and evaluating the practical applicability as, at the moment, we only anticipate the success of this approach for real data.

We hope to explore these exciting problems in the future. So far, we only scratched the surface, but in our opinion joint inference will eventually lead to a new generation of machine learning methods which will be able to perform complicated classification tasks that are intractable at the moment.

A. Algorithms

A.1. Original HTM

Algorithm A.1: `BottomSpatialPooler` training

Data: input vector $\mathbf{x} \in \mathbb{R}^n$

begin
  $novelCoincidence \leftarrow$ **true**
  forall $\mathbf{c} \in \mathcal{C}$ **do** `// iterate over all coincidences`
    if $\|\mathbf{x} - \mathbf{c}\|_2 < minDistance$ **then** `// compare to input`
      $novelCoincidence \leftarrow$ **false**
      break
  if $novelCoincidence \wedge (|\mathcal{C}| < maxCoincidences)$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{x}\}$
end

Algorithm A.2: `BottomSpatialPooler` inference

Data: input vector $\mathbf{x} \in \mathbb{R}^n$

begin
  $sum \leftarrow 0$
  for $i \leftarrow 1$ **to** $|\mathcal{C}|$ **do** `// iterate over all coincidences`
    $d \leftarrow \|\mathbf{x} - \mathbf{c}_i\|_2$ `// calculate distance`
    $y_i \leftarrow \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{\frac{d}{2\sigma^2}}$ `// calculate Gauss-function`
    $sum \leftarrow sum + y_i$
  for $i = 1$ **to** $|\mathcal{C}|$ **do** `// normalize distribution`
    $y_i \leftarrow \frac{y_i}{sum}$
  return $\mathbf{y}$
end

Algorithm A.3: `MidSpatialPoolerNode` preprocessing for training

Data: input vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ one from each child

begin
  for $i \leftarrow 1$ **to** m **do** `// iterate over the parts of the input data.`
    $winnerIndex \leftarrow \arg\max_j(\mathbf{x}^i_j)$ `// determine biggest value`
    for $j \leftarrow 1$ **to** $|\mathbf{x}^i|$ **do** `// set all values to zero`
      $\mathbf{x}^i_j \leftarrow 0.0$
    $\mathbf{x}^i_{winnerIndex} \leftarrow 1.0$ `// set the winning component to one`
  $\mathbf{x} \leftarrow \mathbf{x}^1 \oplus \mathbf{x}^2 \oplus \cdots \oplus \mathbf{x}^m$ `// concatenate the parts`
  return $\mathbf{x}$
end

Algorithm A.4: `MidSpatialPoolerNode` inference

Data: input vector $\mathbf{x} \in \mathbb{R}^n$

begin
  $sum \leftarrow 0$
  for $i \leftarrow 1$ **to** $|\mathcal{C}|$ **do** `// iterate over all coincidences`
    $\mathbf{a} \leftarrow \mathbf{x} \cdot \mathbf{c}_i$ `// multiply input data with coincidence`
    if *poolingType* = SUM **then**
      $y_i \leftarrow \sum_{j=1}^{n} a_j$
    else if *poolingType* = PRODUCT **then**
      $y_i \leftarrow \prod_{j=1}^{n} a_j$
    $sum \leftarrow sum + y_i$
  for $i = 1$ **to** $|\mathcal{C}|$ **do** `// normalize distribution`
    $y_i \leftarrow \frac{y_i}{sum}$
  return $\mathbf{y}$
end

Algorithm A.5: `TemporalPoolerNode` training

`/* Maintain the transition matrix */`

Data: coincidence probability distribution $\mathbf{y} \in \mathbb{R}^{|\mathcal{C}|}$

begin
  $w \leftarrow \arg\max_j \mathbf{y}_j$
  if $w_{old} > 0$ **then**
    $\mathbf{T}[w_{old}, w] \leftarrow \mathbf{T}[w_{old}, w] + 1$
  $w_{old} \leftarrow w$
end

Algorithm A.6: `TemporalPoolerNode` temporal grouping

```
/* Temporal grouping algorithm used by the TemporalPoolerNode        */
Data: Transition matrix T
begin
    for i ← 1 to |C| do                          // normalize transition matrix
        colsum ← 0
        for j ← 1 to |C| do
            colsum ← colsum + T[j, i]
        for j ← 1 to |C| do
            T̂[j, i] ← T[j,i] / colsum
    for i ← 1 to |C| do                                // initialize the groups
        g_i ← {i}
    G ← {g_1, g_2, ..., g_|C|}
    while |G| > maxGroups do  // merge groups until maxGroups is reached
        i, j ← arg max_{i,j∈{1,...,|G|}}(connectivity(g_i, g_j, T̂))
        g_i ← g_i ∪ g_j
        G ← G \ g_j
end
```

Algorithm A.7: `TemporalPoolerNode` connectivity

```
/* calculate connectivity between two groups                          */
Data: groups g_1 and g_2, normalized transition matrix T̂
begin
    n ← |g_1| + |g_2|
    m ← max_{i∈g_1, j∈g_2}(max(T̂[i, j], T̂[j, i]))
    return m / n
end
```

Algorithm A.8: `TemporalPoolerNode` inference

```
/* calculate the feed-forward data of the temporal pooler             */
Data: coincidence probability distribution y ∈ R^|C|
begin
    for i ← 1 to |G| do
        x_i ← max_{j∈g_i}(c_j)
    return x
end
```

Algorithm A.9: `SupervisedMapperNode` training

```
/* Maintain the mapping table                                    */
```

Data: coincidence probability distribution $\mathbf{y} \in \mathbb{R}^{|\mathcal{C}|}$ and class label $c \in \mathbb{N}$

begin
  $w \leftarrow \arg\max_j \mathbf{y}_j$
  $\mathbf{M}[w, c] \leftarrow \mathbf{M}[w, c] + 1$
end

Algorithm A.10: `SupervisedMapperNode` inference

```
/* Determine the best class label                                */
```

Data: coincidence probability distribution $\mathbf{y} \in \mathbb{R}^{|\mathcal{C}|}$

begin
  $w \leftarrow \arg\max_i \mathbf{y}_i$
  $c \leftarrow \arg\max_i \mathbf{M}[w, i]$
  return c
end

A.2. Improved Algorithms

Algorithm A.11: `QSpatialPoolerNode` prepareFeedback

```
/* Translate the feedback                                        */
```

Data: coincidence probability distribution $\boldsymbol{\psi} \in \mathbb{R}^{|\mathcal{C}|}$

begin
  for $i \leftarrow 1$ **to** $|\mathcal{C}|$ **do**
    $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} + \mathbf{c}_i \cdot \psi_i$
  $\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_m \leftarrow split(\boldsymbol{\pi})$
  return $\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_m$
end

Algorithm A.12: `QSpatialPoolerNode` split

```
/* Split a pattern distribution into one part per child.          */
```
Data: pattern distribution $\mathbf{p} \in \mathbb{R}^{|\mathbf{x}|}$

begin
- $start \leftarrow 1$
- **for** $i \leftarrow 1$ **to** m **do**
  - **for** $j \leftarrow 0$ **to** $|\mathbf{x}_i| - 1$ **do**
    - $\pi_{i,j} = p_{start+j}$
    - $start \leftarrow start + 1$
- **for** $i \leftarrow 1$ **to** m **do** `// normalize the parts`
  - $\boldsymbol{\pi}_i = \frac{\boldsymbol{\pi}_i}{\sum_{j=1}^{|\boldsymbol{\pi}_i|} \pi_{i,j}}$
- **return** $\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_m$

end

Algorithm A.13: `QSpatialPoolerNode` unfold

```
/* unfold given coincidence                                       */
```
Data: coincidence index $i_c \in \mathbb{N}$ and timestamp t

begin
- $\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_m \leftarrow split(\mathbf{c}_{i_c})$
- $channels \leftarrow \{\}$
- **for** $i \leftarrow 1$ **to** m **do**
  - $channels \leftarrow channels \cup \text{child}_i.\text{unfold}(\arg\max_{j=1}^{|\boldsymbol{\pi}_i|}(\pi_{i,j}), t)$
- **return** $channels$

end

Algorithm A.14: `QSequencerNode` train

```
/* train PPM predictor and store winner sequence                  */
```
Data: input data $\mathbf{y} \in \mathbb{R}^{|\mathcal{C}|}$

begin
- $w \leftarrow \arg\max_{i=0}^{|\mathbf{y}|}(y_i))$
- $seq \leftarrow seq \oplus w$
- PPM.$train(w)$

end

Algorithm A.15: `QSequencerNode` switchToInference

```
   /* split training sequence into                                          */
   Data: training sequence seq and trained predictor PPM
 1 begin
 2     for i ← 1 to m do                  // create the prediction entropy sequence
 3         context ← [seq_{i−k}, seq_{i−k+1}, ..., seq_i]
 4         p ← PPM.predict(context)
 5         H_i ← −Σ_{j=1}^{|p|} (p_j · log_{|p|} p_j)
 6     H_avg ← (Σ_{i=1}^{|H|} H_i) / |H|
 7
 8     S ← {}
 9     b ← 1
10     for i ← 1 to m do                                  // form sequences
11         if H_i > H_avg then
12             s_new = [seq_b, seq_{b+1}, ..., seq_{b+k−1}]
13             S ← S ∪ {s_new}
14             b ← i + 1
15     return S
16 end
```

Algorithm A.16: `QSequencerNode` prepareFeedForwardData

```
/* maintain short-term memory and prepare feed-forward data if entropy
   is high                                                          */
Data: coincidence probability distribution y^t ∈ ℝ^|𝒞|
begin
    STM ← STM ⊕ y^t
    if (|STM| > k) ∧ (b > 1) then                  // remove unnecessary memories
        i_cut ← min(|STM| − k, b)
        STM ← STM.subsequence(i_cut, |STM|)
        b ← b − i_cut
    if entropy > H_avg then                              // if new sequence starts
        divisionMemory ← divisionMemory ⊕ t
    p ← PPM.predict(STM.subsequence(|STM| − k, |STM|))
    entropy ← −Σ_{j=1}^{|p|} (p_j · log_{|p|} p_j)
    if entropy > H_avg then                          // if next symbol is uncertain
        for i ← 1 to |𝒮| do                      // calculate sequence distribution
            x_i ← Π_{j=1}^{|s_i|} STM[b + i − 1]_{s_i,j}
        b ← |STM| + 1
    return x
end
```

Algorithm A.17: `QSequencerNode` prepareFeedback

```
/* prepare the feedback                                             */
Data: sometimes receives feeback π from parent
begin
    if entropy <= H_avg then          // if entropy is high return prediction
        return p
    else                          // otherwise translate feedback from parent
        for i ← 1 to |𝒮| do
            symbol ← s_{i,1}
            ψ_symbol ← ψ_symbol + π_i
        ψ ← (Σ_{i=1}^{|ψ|} ψ_i) / |ψ|                        // normalize distribution
        return ψ

end
```

Algorithm A.18: `QSequencerNode` unfold

```
   /* unfold given sequence                                                    */
   Data: sequence index i_s ∈ ℕ and timestamp t
1  begin
2      unfoldLength ← min_{t_end ∈ divisionMemory}(t_end > t)
3      channels ← {}
4      j ← 1
5      t_start ← t
6      while |channels_1| < unfoldLength do   // while unfoldLength not reached
7          ch ← child.unfold(s_{i_s,j}, t_start)          // unfold coincidences
8          for j ← 1 to |channels| do                     // and append channels
9              channels_j ← channels_j ⊕ ch_j
10         j ← j + 1
11         t_start ← t_start + |ch_1|
12     return channels
13 end
```

B. Nomenclature

$\mathbf{1}_i(j)$ Indicator function which is 1 if $i = j$ and 0 otherwise

$\mathbf{T}$ Transition matrix used by the Temporal Pooler. $\mathbf{T} = (T_{ij}) \in \mathbb{M}(|\mathcal{C}|, |\mathcal{C}|)$

$\mathbf{x}^t$ Both, the feed-forward data of the Sequencer, and the input data of the parent Spatial Pooler. Can also be the input data for the bottom-layer. $\mathbf{x}^t \in \mathbb{R}^{|\mathcal{S}|}$

$\mathcal{S}$ Set of sequences of the current sequencer node. $\mathcal{S} \in$

σ Standard deviation of the normal distribution used by the Spatial Pooler. $\sigma \in \mathbb{R}$

$H(A)$ Entropy of discrete random variable A.

k Order of the Markov-Model. $k \in \mathbb{N}$

$maxCoincidences$ Parameter of the Spatial Pooler. $|\mathcal{C}| \leq maxCoincidences$

$maxDistance$ Parameter of the Spatial Pooler

$maxGroups$ Parameter of the Temporal Pooler. $|\mathcal{G}| \leq maxGroups$

N_{ij} The j-th node of the i-th layer.

$P(A)$ Probability for event A.

$\boldsymbol{\pi}_m^t$ Feedback from the Spatial Pooler to the m-th child at timestep t

$\boldsymbol{\psi}^t$ Feedback from the Sequencer to it's child at timestep t.

$\mathcal{C}$ Set of stored coincidences. $\mathcal{C} \in 2^{\mathbb{R}^n}$

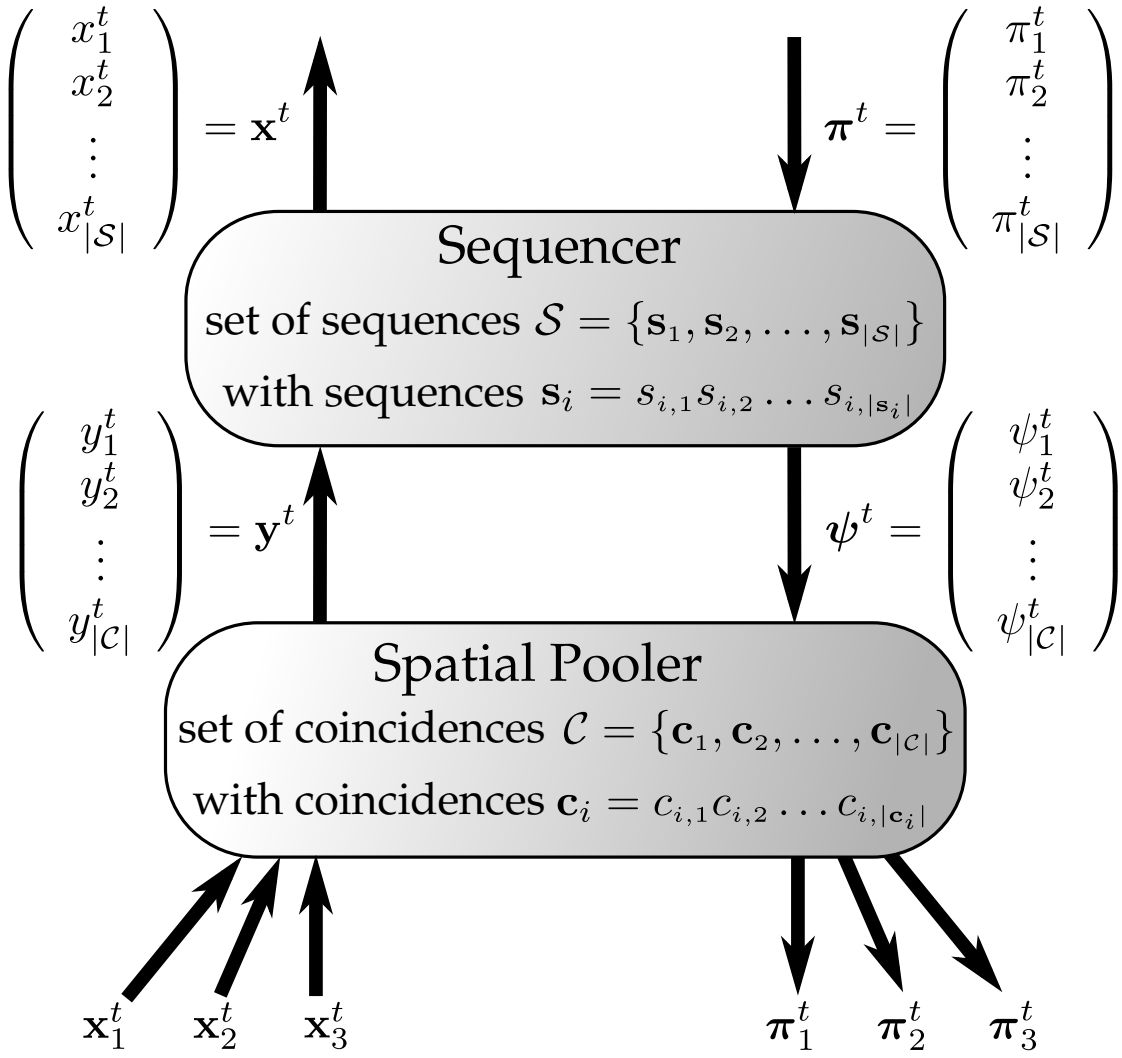


Figure B.1.: Overview over the used notation.

Bibliography

S. Becker and G.E. Hinton. Self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, 355(6356):161–163, 1992. URL http://sciwebserver.science.mcmaster.ca/Psychology/becker/papers/BeckerHintonNature92.pdf.

R. Begleiter, R. El-Yaniv, and G. Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22(1):385–421, 2004. URL http://biozon.org/people/golan/papers/VMMs.pdf.

Dave Benson. *Music: A Mathematical Offering*. Cambridge University Press, 2006. URL http://www.maths.abdn.ac.uk/~bensondj/html/music.pdf.

Bruce A. Bobier and Michael Wirth. Content-based image retrieval using hierarchical temporal memory. In Abdulmotaleb El-Saddik, Son Vuong, Carsten Griwodz, Alberto Del Bimbo, K. Selçuk Candan, and Alejandro Jaimes, editors, *ACM Multimedia*, pages 925–928. ACM, 2008. ISBN 978-1-60558-303-7. URL http://dblp.uni-trier.de/db/conf/mm/mm2008.html#BobierW08.

Jake Bouvrie, Lorenzo Rosasco, and Tomaso Poggio. On invariance in hierarchical models. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22, pages 162–170, 2009. URL http://books.nips.cc/papers/files/nips22/NIPS2009_0978.pdf.

R.A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991. URL http://www.ai.mit.edu/people/brooks/papers/representation.pdf.

H.L. Chieu, W.S. Lee, and L.P. Kaelbling. Activity recognition from physiological data using conditional random fields. In *SMA Symposium. Singapore-MIT Alliance*. Citeseer, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.49&rep=rep1&type=pdf.

Edsger Wybe Dijkstra. The notational conventions i adopted, and why. Website, 2000. URL http://userweb.cs.utexas.edu/users/EWD/index13xx.html.

M. Feder and N. Merhav. Relations between entropy and error probability. *Information Theory, IEEE Transactions on*, 40(1):259–266, 2002. doi: 10.1109/18.272494. URL http://webee.technion.ac.il/people/merhav/papers/entropyerrprob.pdf.

Shai Fine, David Haussler, and Naftali Tishby. The hierarchical hidden markov model: Analysis and applications. In *Machine Learning*, volume 32, pages 41–62. Springer,

July 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.7940&rep=rep1&type=pdf.

S. Garalevicius. Analysis and implementation of the memory-prediction framework. Website, May 2005a. URL http://www.phillylac.org/prediction/2005%2005%20Analysis%20and%20Implementation%20of%20MPF.pdf.

S. Garalevicius. Using memory-prediction framework for invariant pattern recognition. Website, December 2005b. URL http://phillylac.org/prediction/2005%2012%20Using%20MPF%20for%20Pattern%20Recognition.pdf.

Saulius J. Garalevicius. Memory prediction framework for pattern recognition: Performance and suitability of the bayesian model of visual cortex. Technical report, Department of Computer and Information Sciences, Temple University, 2007. URL https://www.aaai.org/Papers/FLAIRS/2007/Flairs07-018.pdf.

Dileep George. *How The Brain Might Work: A Hierarchical And Temporal Model For Learning*. Phd thesis, Stanford University, June 2008. URL http://www.numenta.com/for-developers/education/DileepThesis.pdf.

Dileep George and Jeff Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS Computational Biology*, 5:Issue 10, October 2009. doi: 10.1371/journal.pcbi.1000532. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2749218/pdf/pcbi.1000532.pdf.

Dileep George and Bobby Jaros. The htm learning algorithms. Technical report, Numenta Inc., March 2007. URL http://www.numenta.com/for-developers/education/Numenta_HTM_Learning_Algos.pdf.

F. Golcher. Statistical text segmentation with partial structure analysis. In *Proceedings of KONVENS*, pages 44–51, 2006. URL http://amor.rz.hu-berlin.de/~golcherf/index.pdf.

Rhys Goldstein. On the unification of mathematical notation and programming notation. Website, August 2008. URL http://www.rhysgoldstein.com/Goldstein__Unification.pdf.

A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM, 2006. ISBN 1595933832. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.6306&rep=rep1&type=pdf.

K.K. Gupta, B. Nath, and K. Ramamohanarao. Conditional random fields for intrusion detection. In *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, volume 1, pages 203–208. IEEE, 2007. doi: 10.1109/AINAW.2007.126. URL http://ww2.cs.mu.oz.au/~kgupta/files/papers/2007fina.pdf.

Yensy James Hall and Ryan E. Poplin. Using numenta's hierarchical temporal memory to recognize captchas. Website, 2007. URL `http://www.pembrokeballet.com/10701-HTM_CAPTCHA.pdf`.

Josh Hartung, Jay McCormack, and Frank Jacobus. Support for the use of hierarchical temporal memory systems in automated design evaluation: A first experiment. In *Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, volume 8, pages 853–862, 2009. doi: 10.1115/DETC2009-87702. URL `http://www.webpages.uidaho.edu/~mccormack/papers/DETC2009-87702.pdf`.

J. Hawkins, D. George, and J. Niemasik. Sequence memory for prediction, inference and behaviour. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 364(1521):1203–1209, May 2009. doi: 10.1098/rstb.2008.0322. URL `http://rstb.royalsocietypublishing.org/content/364/1521/1203.full.pdf+html`.

Jeff Hawkins and Sandra Blakeslee. *On Intelligence*. Holt Paperback, 1. edition, 2005. URL `http://www.scribd.com/doc/2942162/HawkinsJeff-On-Intelligence?autodown=pdf`.

Jeff Hawkins and Dileep George. Hierarchical temporal memory concepts, theory, and terminology. Technical report, Numenta Inc., 2006. URL `http://www.numenta.com/Numenta_HTM_Concepts.pdf`.

Helge Homburg, Ingo Mierswa, Bülent Möller, Katharina Morik, and Michael Wurst. A benchmark dataset for audio classification and clustering, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.334&rep=rep1&type=pdf`.

Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, April 2010. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

Mohammed Waleed Kadous. *Temporal Classification: Extending the Classification Paradigm to Multivariate Time Series*. Phd thesis, The University of New South Wales, School of Computer Science and Engineering, October 2002. URL `http://www.cse.unsw.edu.au/~waleed/phd/`.

Tomasz Kapuscinski. Using hierarchical temporal memory for vision-based hand shape recognition under large variations in hand's rotation. In Leszek Rutkowski, Rafal Scherer, Ryszard Tadeusiewicz, Lotfi A. Zadeh, and Jacek M. Zurada, editors, *ICAISC (2)*, volume 6114 of *Lecture Notes in Computer Science*, pages 272–279. Springer, 2010. ISBN 978-3-642-13231-5. URL `http://dblp.uni-trier.de/db/conf/icaisc/icaisc2010-2.html#Kapuscinski10`.

R. Klinger and K. Tomanek. Classical probabilistic models and conditional random fields. *Technische Universit*
"at Dortmund, Electronic Publication, TR07-2-013:1–31, 2007. URL `http://ls11-www.informatik.uni-dortmund.de/_media/techreports/tr07-13.pdf`.

J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pages 282–289. Citeseer, 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.9849&rep=rep1&type=pdf`.

H. Lee, R. Grosse, R. Ranganath, and A.Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, volume 382, pages 609–616. ACM, 2009a. doi: 10.1145/1553374.1553453. URL `http://portal.acm.org/ft_gateway.cfm?id=1553453&type=pdf&coll=GUIDE&dl=GUIDE&CFID=100604392&CFTOKEN=40133546`.

Honglak Lee, Peter Pham, Yan Largman, and Andrew Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, chapter 1096–1104. Neural Information Processing Systems Foundation, 2009b. URL `http://ai.stanford.edu/~ang/papers/nips09-AudioConvolutionalDBN.pdf`.

Y. Liu, J. Carbonell, P. Weigele, and V. Gopalakrishnan. Segmentation conditional random fields (SCRFs): A new approach for protein fold recognition. In *Research in Computational Molecular Biology*, pages 408–422. Springer, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.1562&rep=rep1&type=pdf`.

J.B. Maxwell, P. Pasquier, and A. Eigenfeldt. Hierarchical sequential memory for music: A cognitive model. In *10th International Society of Music Information Retrieval Conference*, pages 429–434, October 2009a. URL `http://www.rubato-music.com/home/JBM_Research_files/JBM_HSMM_2009%20-%20Revised.pdf`.

J.B. Maxwell, P. Pasquier, and A. Eigenfeldt. Hierarchical sequential memory for music: A cognitively-inspired approach to generative music. Website, 2009b. URL `http://www.rubato-music.com/home/JBM_Research_files/HSMM_813.pdf`.

J.B. Maxwell, P. Pasquier, and A. Eigenfeldt. The hierarchical sequential memory for music: A cognitively-inspired model for music learning and composition. In preparation for upcoming conference, 2010.

Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. Java package, 2002. URL `http://mallet.cs.umass.edu`.

J. W. Miller and P. H. Lommel. Biomimetic sensory abstraction using hierarchical quilted self-organizing maps. In *PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*, volume 6384, October 2006. doi: 10.1117/12.686183. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.1401&rep=rep1&type=pdf`.

H. Mobahi, R. Collobert, and J. Weston. Deep learning from temporal coherence in video. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 737–744. ACM, 2009. URL `http://portal.acm.org/citation.cfm?id=1553469`.

Craig G. Nevill-Manning. *Inferring Sequential Structure*. Phd thesis, University of Waikato, May 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.9964&rep=rep1&type=pdf`.

Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997. URL `http://www.eecs.harvard.edu/~michaelm/CS222/sequitur.pdf`.

Numenta. *Zeta1 Algorithms Reference*. Numenta Inc., 1.5 edition, August 2007a. Shipped with the NuPIC 1.5 release.

Numenta. *Hierarchical Temporal Memory Comparison with Existing Models*. Numenta Inc., 1.0.1 edition, 2007b. URL `http://www.numenta.com/for-developers/education/HTM_Comparison.pdf`.

Numenta. *Problems that Fit HTM*. Numenta Inc., 1.0 edition, March 2007c. URL `http://www.numenta.com/for-developers/education/ProblemsThatFitHTMs.pdf`.

Numenta. *Getting Started With NuPIC*. Numenta Inc., September 2008a. URL `http://www.numenta.com/for-developers/software/pdf/nupic_gettingstarted.pdf`.

Numenta. *Numenta Platform for Intelligent Computing Node Plugin Developer's Guide*. Numenta Inc., 1.2.1 edition, June 2008b. URL `http://www.numenta.com/for-developers/software/pdf/nupic_plug_guide.pdf`.

Numenta. *Advanced NuPIC Programming*. Numenta Inc., 1.8.1 edition, September 2008c. URL `http://www.numenta.com/for-developers/software/pdf/nupic_prog_guide.pdf`.

Marco Ramoni Paola Sebastiani and Paul Cohen. Sequence learning via bayesian clustering by dynamics. In Ron Sun and C. Lee Giles, editors, *Sequence Learning*, volume 1828 of *Lecture Notes in Artificial Intelligence*, pages 11–34. Springer, 2001. doi: 10.1007/3-540-44565-X_2.

A.J. Perea, J.E. Meroño, and M.J. Aguilera. Application of numenta hierarchical temporal memory for land-use classification. *South African Journal of Science*, 105(9-10):370–375, October 2009. URL `http://www.scielo.org.za/pdf/sajs/v105n9-10/a1410510.pdf`.

Rafael Coimbra Pinto. *A Neocortex Inspired Hierarchical Spatio-Temporal Pattern Recognition System*. Bachelor thesis, Universidade Federal Do Rio Grande Do Sul, June 2009. URL `http://www.inf.ufrgs.br/~rcpinto/tcc/monografia.pdf`.

L. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE ASSp Magazine*, 3(1 Part 1):4–16, 1986. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?&arnumber=1165342`.

L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Februrary 1989.

Kiruthika Ramanathan, Luping Shi, Jianming Li, Kian Guan Lim, Ming Hui Li, Zhi Ping Ang, and Tow Chong Chong. A neural network model for a hierarchical spatio-temporal memory. In Mario Köppen, Nikola K. Kasabov, and George G. Coghill, editors, *ICONIP (1)*, volume 5506 of *Lecture Notes in Computer Science*, pages 428–435. Springer, 2008. ISBN 978-3-642-02489-4. URL `http://dblp.uni-trier.de/db/conf/iconip/iconip2009-1.html#RamanathanSLLLAC09`.

D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine learning*, 25(2):117–149, 1996. doi: 10.1023/A:1026490906255. URL `http://www.springerlink.com/index/U43U01836770W366.pdf`.

David Rozado, Francisco B. Rodriguez, and Pablo Varona. Optimizing hierarchical temporal memory for multivariable time series. In Konstantinos I. Diamantaras, Wlodek Duch, and Lazaros S. Iliadis, editors, *ICANN (2)*, volume 6353 of *Lecture Notes in Computer Science*, pages 506–518. Springer, 2010. ISBN 978-3-642-15821-6. URL `http://dblp.uni-trier.de/db/conf/icann/icann2010-2.html#RozadoRV10`.

Federico Sassi, Luca Ascari, and Stefano Cagnoni. Classifying human body acceleration patterns using a hierarchical temporal memory. In Roberto Serra and Rita Cucchiara, editors, *AI*IA*, volume 5883 of *Lecture Notes in Computer Science*, pages 496–505. Springer, 2009. ISBN 978-3-642-10290-5. URL `http://dblp.uni-trier.de/db/conf/aiia/aiia2009.html#SassiAC09`.

Nathan C. Schey. Song identification using the numenta platform for intelligent computing. Bachelor thesis, Department of Computer Science and Engineering at The Ohio State University, May 2008. URL `https://kb.osu.edu/dspace/bitstream/1811/32025/1/Schey_Thesis.pdf`.

M. Slaney. Auditory toolbox. Technical report, Apple Computer Company, 1993.

Ron Sun. Introduction to sequence learning. In Ron Sun and C. Lee Giles, editors, *Sequence Learning*, volume 1828 of *Lecture Notes in Artificial Intelligence*, pages 1–10. Springer, 2001. doi: 10.1007/3-540-44565-X_1. URL `http://www.springerlink.com/content/knnuaeb394xlxcu0/fulltext.pdf`.

C. Sutton and A. McCallum. An introduction to conditional random fields for relational learning. In L. Getoor and B. Taskar, editors, *Introduction to statistical relational learning*, chapter 4, pages 93–126. The MIT Press, 2007. URL http://www.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf.

John Thornton, Torbjorn Gustafsson, Michael Blumenstein, and Trevor Hine. Robust character recognition using a hierarchical bayesian network. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.5326&rep=rep1&type=pdf.

Laura Firoiu Tim Oates and Paul Cohen. Using dynamic time warping to bootstrap hmm-based clustering of time series. In Ron Sun and C. Lee Giles, editors, *Sequence Learning*, volume 1828 of *Lecture Notes in Artificial Intelligence*, pages 35–52. Springer, 2001. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.3394&rep=rep1&type=pdf.

Joost van Doremalen and Lou Boves. Spoken digit recognition using a hierarchical temporal memory. In *Proceedings Interspeech*, pages 2566–2569, 2008. doi: 10.1117/12.686183. URL http://lands.let.ru.nl/acorns/documents/publications/Interspeech-2008/Doremalen-Boves.pdf.

DeLiang Wang. Anticipation model for sequential learning of complex sequences. In Ron Sun and C. Giles, editors, *Sequence Learning*, volume 1828 of *Lecture Notes in Artificial Intelligence*, pages 53–79. Springer, 2001. doi: 10.1007/3-540-44565-X_4. URL http://dx.doi.org/10.1007/3-540-44565-X_4.

Brandyn Jerad Webb. Fusion-reflection: Self-supervised learning, 1993. URL http://www.sifter.org/~brandyn/Furf-I.pdf.

G.P. Zhang. Neural networks for classification: a survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 30(4):451–462, 2002. ISSN 1094-6977. URL http://www-vis.lbl.gov/~romano/mlgroup/papers/neural-networks-survey.pdf.

B. Zupan, M. Bohanec, J. Demsar, and I. Bratko. Learning by discovering concept hierarchies. *Artificial Intelligence*, 109(1):211–242, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.2479&rep=rep1&type=pdf.