Faculteit
Letteren

Vakgroep
Taal- en Spraaktechnologie

RADBOUD
UNIVERSITEIT
NIJMEGEN

Hierarchical Temporal Memory Networks for Spoken Digit Recognition

Joost van Doremalen

Begeleiding:
Prof. Dr. Lou Boves

2006 – 2007

Abstract

Contents

Chapter 1

Introduction

Recently a new theory on brain function has been presented in [1]. The main tenets of this new, still immature, theory can be modelled using Bayesian techniques. This model is called a *Hierarchical Temporal Memory* or HTM network. In these networks, spatial and temporal relations are learned in a hierarchical architecture. The result of this learning process is essentially a Bayesian network with an inverted tree structure. Given a new pattern, the recognition process can be viewed as choosing the high-level belief that best predicts the pattern. This method is currently successfully applied to the recognition of simple images [2]. The system shows invariance across several transformations and is robust with respect to noisy patterns. In this research we intend to apply the concept of HTM networks to the recognition of spoken words and utterances. To that goal we will build and test a system that learns to recognize the 11 words in the TIDIGITS database [3].

1.1 Objectives

The conventional use of *Hidden Markov Modelling* (HMM) methods in ASR seem to be insufficient for reaching the goals of researchers in this area [4]. HMM approaches are incapable of dealing with the unexpected events that abound in natural speech. Therefore, the generalization capabilities of the model presented in [2] also seem interesting for automatic speech recognition (ASR) systems, because these systems have to be speaker-invariant and robust to

background noise.

In this research, we will address the problem of spoken digit recognition from the perspective of the memory-prediction framework. More concrete, this means we will implement an HTM network which is able to recognize spoken digits. The input will be a feature representation of the already segmented speech signal. Our model will give us a sense of the capabilities of the memory-prediction framework in implemented ASR systems. Hopefully, this will lead to research that can result in more robust ASR systems.

1.2 Methodology

First, we will extract all utterances of isolated digits from the TIDIGITS database [3] and divide them in proper train- and test sets. For practical reasons, we assume the speech input signal to be already segmented. From these utterances we will build auditory feature representations. Subsequently we will implement an HTM network with the help of existing software tools. We will evaluate whether the system is capable of successfully categorizing the speech signals and apply conventional measures like the proportion of correctly recognized digits in a test set that the system has not seen during training. Furthermore, we will investigate how our system compares to a state-of-the-art HMM model trained on the same data.

1.3 Overview of this thesis

In Chapter 2 we will introduce HTM networks and discuss the algorithms involved. In Chapter 3 we will elaborate on the data we have used in this research, along with the preprocessing steps needed to use this data to train and test our HTM network. The implementation of this network is the topic of Chapter 4. In Chapter 5 we will present the results obtained with this implementation. Furthermore, we will present several experiments with the network its input representations, architecture and parameter settings with the goal of gaining a deeper understanding of HTM networks as well as improving the performance of the network. In Chapter 6 we will present the main conlusions obtained from this research, along with some possible directions for further research.

Chapter 2

Hierarchical Temporal Memory Networks

In this chapter we will introduce HTM networks. We will give an overview of their structure and the learning and inference algorithms involved.

2.1 The Memory-Prediction Framework and Hierarchical Temporal Memory Networks

The memory-prediction framework introduced in [1] gives rise to the HTM network architecture and the learning and inference algorithms. To give some contextual embedding of HTM networks, we will touch upon the main tenets of this framework quickly before delving deeper into HTM network theory.

- **The neocortex groups together diverse spatial patterns occuring frequently close in time.**

  For example, if we see a person running from left to right in our visual field, the low level sensory input changes constantly, but our concept of a running person remains remarkably stable. This example shows that the neocortex uses temporal information to group these diverse spatial patterns together.

- **The neocortex forms complex concepts from simple concepts.**

  In several experiments it is shown that visual pattern recognition in the visual cortex is organized hierarchically [5]. Hierarchically lower visual areas become active in reaction to simple stimuli like lines and corners, while higher areas react to complex stimuli like faces. Apparently, the neocortex uses these simple concepts or features to form increasingly more complex concepts.

- **A common cortical algorithm exists.**

  The neocortex has a relatively homogenous organization: areas dealing with vision, sound, touch etc. look remarkably similar [6]. Based on this fact a common cortical algorithm is assumed. Every part of the cortex (be it visual, auditory or other) functions in a similar way. This also means that every part in the hierarchy works similar, so the visual area recognizing faces operates in the same way as the lower visual area recognizing lines and corners.

In the rest of this thesis, we will not be concerned with further details of the memory-prediction framework and cortical functioning. It is however important to note that this theory is the origin of HTM networks and thus they can be given a biological interpretation.

2.2 Numenta Platform for Intelligent Computing

Numenta (see `http://www.numenta.com` for details) is a company founded by Jeff Hawkins [1] [2] developing and promoting the HTM technology. The Numenta Platform for Intelligent Computing (or NuPIC) developed by Numenta is a set of software tools which allows developers to create HTM network applications on a fairly abstract level. In this research, we will use NuPIC to create, train and test our HTM network. We will now describe the HTM algorithms as implemented in NuPIC version 1.2. It is important to note that currently NuPIC is still in development and that the algorithms hinted upon in [1] are far from being fully implemented. For more information about the current version of NuPIC, see `http://www.numenta.com/for-developers/education/algorithms.php`.

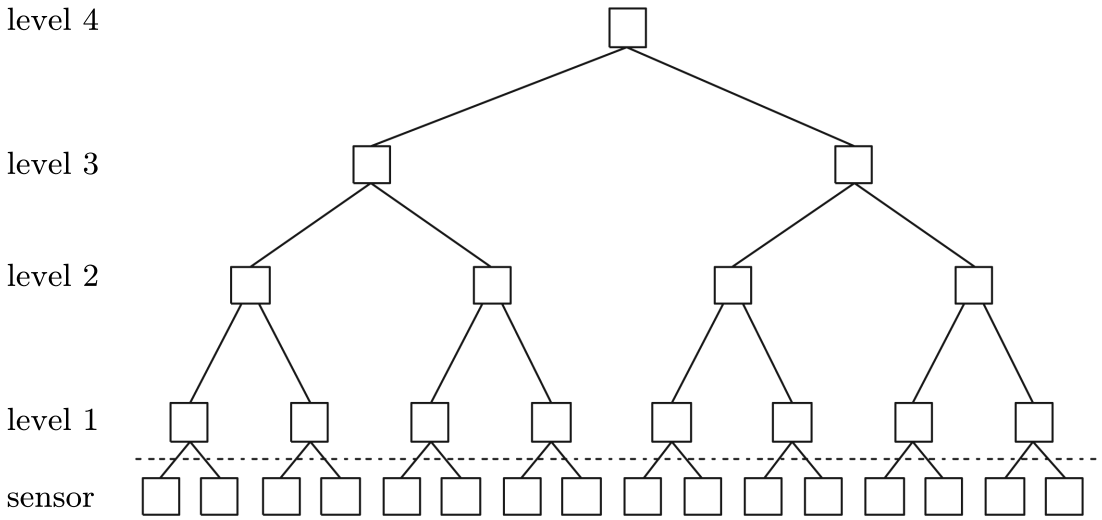

Figure 2.1: *This HTM network consists of four levels of nodes. The boxes beneath the dotted line together represent the 16 elements of the input vectors.*

2.3 Hierarchical Temporal Memory Networks

An HTM network is a collection of linked nodes, organized in a tree-shaped hierarchy. See Fig. 2.1 for an example of an HTM network. HTM networks consist of several layers or *levels* of nodes, with one node at the top level. HTM networks operate in two stages: the *learning* stage and the *inference* stage. During learning stage, the network is exposed to *training patterns* and it builds a model of this data. During inference stage, the network recognizes the new, usually unseen, *test patterns*. More concretely, during a (supervised) learning stage the network learns what pattern belongs to what category, whereafter during inference the network will generate a *belief distribution* over these categories for every new pattern it sees. Belief distributions (represented by *belief vectors*) are a measure of belief the input pattern belongs to one of the categories.

All of the nodes (except the top node used in supervised learning) process information in the same way, so we will now explain the operation of such a node.

2.4 Hierarchical Temporal Memory Network Nodes

An example of a node is shown in Fig. 2.2. Every node consists of two components: a *spatial pooler* and a *temporal pooler*. Understanding the workings of an HTM network node boils down to understanding these two components during both the learning and inference stage.

2.4.1 Operation of nodes during learning

During the learning stage, the spatial pooler learns to map input data to a number of *quantization centers*. The output of the spatial pooler (and input to the temporal pooler) is in terms of its quantization centers and as such can be seen as a preprocessing step for the temporal pooler, simplifying its input. The temporal pooler learns *temporal groups*, which are groups of quantization centers that frequently occur close in time.

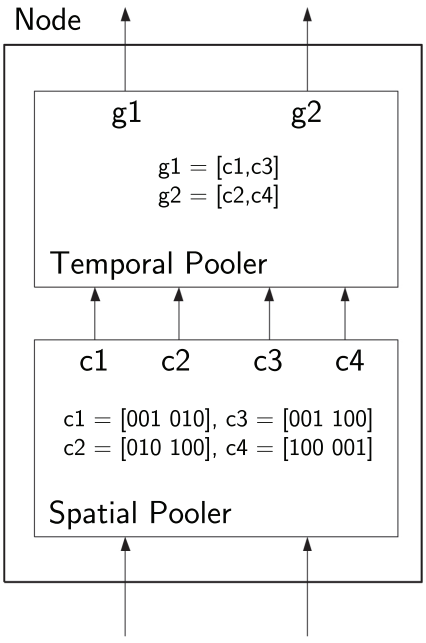

Figure 2.2: *This example of a HTM node has learned four coincidences (c1, c2, c3 and c4) from the output of its child nodes. It has formed two temporal groups (g1 and g2) and it communicates with its parent via a belief distribution over these two groups.*

2.4.1.1 Operation of the spatial pooler during learning

The spatial poolers from nodes at the first level receive raw data from the sensor, the spatial poolers from nodes higher in the hierarchy receive the outputs from child nodes. The input of the spatial poolers of nodes higher in the hierarchy are the concatenations of the output of

their children. The input of the spatial pooler is represented by a row vector and the role of the spatial pooler is to quantize this vector and build a matrix from these quantization centers. This matrix is empty before training. The vectors in this matrix (the quantization centers) are coined *coincidences*, hence the matrix is called a *coincidence matrix*.

Three spatial pooler algorithms exist: *Gaussian*, *Dot* and *Product*. During learning, the Dot and Product algorithms work the same. The Gaussian spatial pooler algorithm is used for nodes at the first level, whereas the nodes higher in the hierarchy use the Dot or Product spatial pooler algorithms. A SPATIALPOOLERALGORITHM parameter specifies which algorithm to use, although it is common to use the same algorithm for every node up the hierarchy. We will now describe how each of these three spatial pooler algorithms operate during learning.

Gaussian spatial pooler The Gaussian spatial pooler algorithm[1], used in first level nodes, compares the raw input vectors to existing coincidences in the coincidence matrix. If the Euclidean distance between this input vector and an existing coincidence is small enough, the input is considered to be an instance of the same coincidence[2] and the count for that coincidence is incremented and stored in memory. The distance between an input vector x and an existing coincidence w is computed as in Eq. 2.1.

$$d^2(\vec{x}, \vec{w}) = \sum_{i=1}^{D} (x_i - w_i)^2 \tag{2.1}$$

where D is the dimensionality of the vectors. The threshold of pooling or not pooling an input vector with an existing coincidence is the parameter MAXDISTANCE. In other words, if $\forall \vec{w}(d^2(\vec{x}, \vec{w}) >$MAXDISTANCE) the input vector $\vec{x}$ is stored as a new coincidence, otherwise it is pooled with the closest existing coincidence. Of course, if MAXDISTANCE is too high, the spatial pooler will store too few coincidences, whereas if it is too low, it will pool too few coincidences together and store too many. It is also worth noting that coincidences, once stored in the coincidence matrix, never change. In Section 4.2.2 we will discuss how to set MAXDISTANCE to a reasonable value.

Dot/Product spatial pooler Because the Dot and Product spatial poolers are always part of a node higher in the hierarchy, they receive the concatenations of the outputs of their

[1] So called because of its behaviour during the inference stage, explained in Section 2.4.2.1
[2] Or: have the same underlying *cause* in the world

child nodes. This vector is divided up into $N_{children}$ portions, which is the number of children of the node. These portions are belief distributions over the temporal groups formed by the child nodes. The Dot and Product spatial pooler set the highest value in each of these $N_{children}$ distributions to 1. The other values are set to 0. These new vectors are stored in the coincidence matrix, and the counts of the coincidences that already exist are incremented.

2.4.1.2 Operation of the temporal pooler during learning

The temporal pooler tries to find groups of coincidences that occur frequently together close in time. To this goal, it builds a first-order time-adjacency matrix, from which after learning can be derived how likely a certain transition between each of the coincidences is.

When a new input vector is presented during training, the spatial pooler represents it as one of its learned coincidences i. The temporal pooler then looks back in history a certain amount of steps in time, which is represented by the parameter transitionMemory. For example, when coincidence j was seen one timestep back, then element (j, i) of the time-adjacency matrix is incremented by the value of transitionMemory. Going further back in time, if coincidence k was seen two timesteps back, (k, i) is incremented by transitionMemory-1. Of course this process stops when transitionMemory steps back in time are made.

If the boolean symmetricTime parameter is set to *true*, the time-adjacency matrix will be symmetric. That is, when element (j, i) is incremented with a certain value, (i, j) is incremented with the same value. This would be a good choice if transitions from coincidence i to j are as likely as transitions from coincidence j to i, but in the speech domain this is obviously not the case.

After the learning stage and before inference, when the time-adjacency matrix is formed, the temporal pooler uses this matrix to create temporal groups. The following algorithm is used for creating these groups:

1. Pick the most frequently seen coincidence i and pool it with a temporal group.
2. Pick the N coincidences C on row i of the time-adjacency matrix which have the highest value and pool them to the same temporal group. N is a parameter coined topNeigh-

BORS and thus governs how many coincidences are added in this step.

3. Repeat step 2 for each of the coincidences i.

It is likely that in step 2 coincidences that are most temporally connected are already part of the same temporal group. When no coincidences can be added or the group size reached MAXGROUPSIZE (if it is not set to infinity) the process is terminated and repeated until no coincidences are left ungrouped.

Additionally, a weight matrix is formed. It has as many rows as temporal groups and as many columns as coincidences. Every element (i, j) represents the row-normalized frequency of coincidence j in group i. During inference, this matrix is used by the **sumProp** algorithm discussed in Section 2.4.2.2.

2.4.1.3 Training the whole network

The nodes are trained from bottom to top. That is, the first level nodes are trained on the training set first, then these nodes are set so to *inference* mode (which we will explain below) and the nodes one level in the hierarchy are trained in a similar way. This process repeat until the top node is reached. This node is trained differently. It has no temporal pooler but a *supervised mapper*, which we will explain in the next section.

2.4.1.4 Operation of the top node during learning

A top node consists of a spatial pooler and a supervised mapper. An example of a top node is depicted in Fig. 2.3. For every training input pattern, the supervised mapper receives two inputs during learning: the coincidence from the spatial pooler and the category of the input vector from the category sensor. It has a mapping matrix, which stores how many times a coincidence i belongs to a category c by incrementing element (c, i) everytime it receives these inputs together.

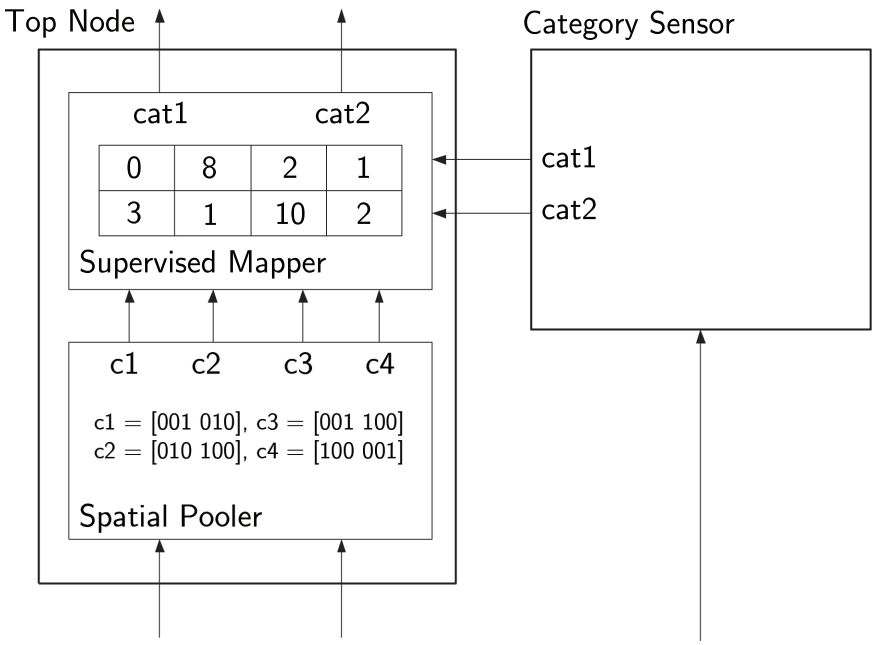

Figure 2.3: *This HTM top node has two inputs: the output from two child nodes and the output from a category sensor. The spatial pooler has learned four coincidences (*c1*, *c2*, *c3* and *c4*) and communicates the coincidence to the supervised mapper. The category sensor communicates one of the two categories to the supervised mapper. Together these two inputs form a mapping matrix in which the elements represent how many times a particular coincidence is seen together with a particular category. For example, the second coincidence* c2 *is seen eight times together with the first category* cat1*. The output of this top node during inference is a belief distribution over the two categories.*

2.4.2 Operation of nodes during inference

After training a node, it is set to *inference* mode. When the whole network is trained, all nodes are in inference state and the network is able to perform inference on new input patterns. At the top, it will generate a belief distribution over the categories it has seen during the learning stage.

2.4.2.1 Operation of the spatial pooler during inference

The three spatial pooler algorithms **Gaussian**, **Dot** and **Product** all work differently during inference stage, but they all convert an input vector into a belief vector over coincidences. As stated before, the Gaussian spatial pooler algorithm is used in first level nodes and the Dot or Product algorithm is used in the nodes higher in the hierarchy.

Gaussian spatial pooler In the Gaussian spatial pooler algorithm, the distance between an input vector x and every one of the learned coincidences is computed using Eq. 2.1. This distance is converted into a belief vector by seeing x as a random sample drawn from a set of multi-dimensional Gaussian distributions all centered on one of the learned coincidences. All these probability distributions have the same variance uniform across all dimensions, governed by the parameter SIGMA or σ, which is the square root of the variance. In Section 4.2.2 we will discuss how to set SIGMA to a reasonable value. Each element i of the belief vector b representing the belief that the input vector x has the same cause as coincidence i, is computed using Eq. 2.2.

$$y_i = \exp\left\{-\frac{d^2(x, W_j)}{2\sigma^2}\right\} \tag{2.2}$$

where d^2 is defined in Eq. 2.1 and W_j is the jth coincidence in the coincidence matrix W.

Dot spatial pooler The Dot spatial pooler algorithm transformed each of the coincidences it saw during training into a coincidence with as many ones as children and the rest zeroes. It takes the dot product of the input vector with each of these learned coincidences, and these result in the elements of the belief vector.

Product spatial pooler The Product spatial pooler algorithm portions the input vector into the outputs of each of its children, takes the dot product with the same portions of the coincidence and then calculates the product of these numbers to give one element of the belief vector over the coincidences.

2.4.2.2 Operation of the temporal pooler during inference

During inference, the temporal pooler receives a belief vector over coincidences from the spatial pooler. It will then calculate a belief distribution over groups. A choice has to be made between two different temporal pooler algorithms during inference: **maxProp** and **sumProp**, governed by the parameter TEMPORALPOOLERALGORITHM.

maxProp The belief in a certain temporal group g with coincidences C is calculated by taking the maximum of the beliefs in coincidences C, derived from the belief vector received from

the spatial pooler. Together these beliefs form the belief vector over temporal groups.

sumProp In the **sumProp** algorithm, the weights matrix formed during learning is used in the calculation of the belief vector over temporal groups. If we call the weights matrix W and the input vector x, then the belief vector $\vec{b}$ is computed using Eq. 2.3.

$$b_i = \sum_j W_{ij} x_j \tag{2.3}$$

2.4.2.3 Operation of the top node during inference

During inference, the spatial pooler of the top node works as described above. The supervised mapper receives a belief vector over coincidences from the spatial pooler and a category from the category sensor. It calculates a belief distribution over these categories. A choice has to be made between two different temporal pooler algorithms during inference: **maxProp** and **sumProp**, governed by the parameter mapperAlgorithm.

maxProp The original mapping matrix formed in the supervised mapper during learning is transformed. In each column, the highest frequency of co-occurences of a particular coincidence and category is set to 1, the rest of the values in that column are set to to 0. If we call this new matrix C and the input vector $\vec{y}$, then the belief vector $\vec{b}$ is computed using Eq. 2.4.

$$b_i = \max_j (C_{ij} y_j) \tag{2.4}$$

where C_{ij} is element (i, j) of C.

sumProp In the **sumProp** algorithm, the mapping matrix is column-normalized. If we call this new matrix C and the input vector $\vec{y}$, then the belief vector $\vec{b}$ is computed using Eq. 2.5

$$b_i = \sum_j C_{ij} y_j \tag{2.5}$$

2.5 Hierarchical Temporal Memory Network and Belief Propagation

A trained HTM network can be interpreted as a graphical model which represents a set of random variables and their independencies. In HTM networks, these random variables are the representations formed at every node. The input pattern of the model can be seen as *evidence*. Using a mechanism called *Belief Propagation* (BP), the network propagates this evidence and alters the *belief states* or degrees of belief in each node's coincidences and temporal groups. At the top of the network this ultimately leads to a belief distribution over the categories given the evidence.

The rest of this thesis deals with our HTM network implemented on the basis of the principles and algorithms discussed in this chapter. The next step is the preprocessing of the data, discussed in the next chapter.

Chapter 3

Materials

In this chapter, we will discuss the data we have used in our research, along with the preprocessing steps needed to obtain input representations for our HTM network.

3.1 Test- en Train data

The dataset used in this research is obtained from the TIDIGITS database [3]. This database contains utterances of isolated digits and connected digit sequences. We extracted the utterances of isolated digits, which comprise 11 words: 'one','two',...,'nine' and 'oh' and 'zero'. All data are sampled at 8kHz and already separated in proper train and test sets. The train set consists of 2420 utterances equally divided between 55 female and 55 male speakers. The test set consists of 1144 utterances equally divided between 52 female and 52 male speakers. Speakers in train and testset are different and all speakers uttered each digit two times, giving 22 utterances per speaker.

3.2 Auditory Preprocessing

We produced auditory feature matrices from the raw signal to train and test our network. Fundamentally, HTM networks are a model of the neocortex. Therefore, we want our auditory feature representations to have some form of physiological and psychological validity. At the

least, the feature representations must be a tonotopical mapping of the speech signal. That is, sounds which are close to each other in frequency must be represented in topologically neighbouring features. This is necessary to infer spatial relations apparent in the pattern. One model with such properties is presented by Patterson & Holdsworth in [7]. This model is backed by empirical findings and therefore a proper candidate for preprocessing our data. After processing our data with the Patterson-Holdsworth filterbank, some other processing steps are carried out to lower the amount of data and scale the data in time. All preprocessing is done with MATLAB. We will now discuss the following preprocessing steps in more detail:

- Patterson-Holdsworth Filterbank
  - Gammatone Auditory Filterbank
  - Meddis' Inner Hair Cell Model
- Decimation
- Time scaling
- Quantization

1. **Gammatone Auditory Filterbank**

   Our gammatone filterbank consists of 32 gammatone filters. The gammatone filter is defined in the time domain with the impulse response given in Eq. 3.1.

$$g(t) = at^{n-1}\cos(2\pi f_c t + \phi)e^{-2\pi bt}(t > 0) \tag{3.1}$$

   where f_c is the centre frequency (in Hz), a and b determine the duration of the impulse response and n is the filter order. Eq. 3.1 resembles the impulse response of human cochlea filters [8] [9]. The filters are equally placed on the ERB or *Equivalent Rectangular Bandwidth* scale from 100Hz to 4000Hz (half the sampling rate). The ERB is a nonlinear rescaling in the frequency domain designed to resemble human frequency selectivity [10] and is (at moderate sound levels) defined by Eq. 3.2.

$$\mathrm{ERB}(f_c) = 24.7\left(\frac{4.37 f_c}{1000} + 1\right) \tag{3.2}$$

   where f_c is the centre frequency (in Hz) and $\mathrm{ERB}(f)$ is the equivalent rectangular bandwidth (in Hz) for that frequency. Clearly, the bandwidth increases from lower to higher

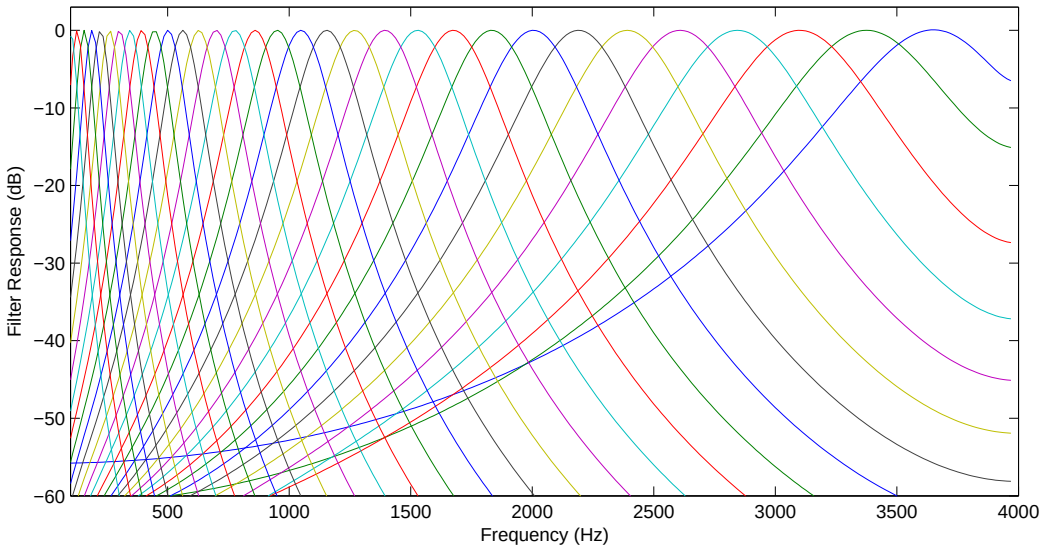

Figure 3.1: *The frequency response of each of the 32 gammatone filters in the filterbank.*

frequencies, which also can be observed in the human auditory system. The frequency response of each filter in the filterbank is shown in Fig. 3.1.

To implement the filterbank we used a Matlab toolbox (the Auditory Toolbox [11]).

2. **Meddis' Inner Hair Cell Model**

   Meddis' Inner Hair Cell Model [12] simulates the mechanical to neural transduction in an inner hair cell on the basilar membrane of the cochlea. The output of the model is the spike probability of such a hair cell. The model accounts for the absolute refractory period and adaptation of these hair cells. The absolute refractory period refers to the period a neuron cannot spike after the previous spike. Adaptation refers to the decrease in spiking freqency (to a certain level) after hair cells are stimulated for a longer period of time. This is caused by a shortage of neurotransmitter between the hair cell and the auditory nerve.

   This model is also implemented in the Auditory Toolbox and each of the 32 filter outputs is processed with it.

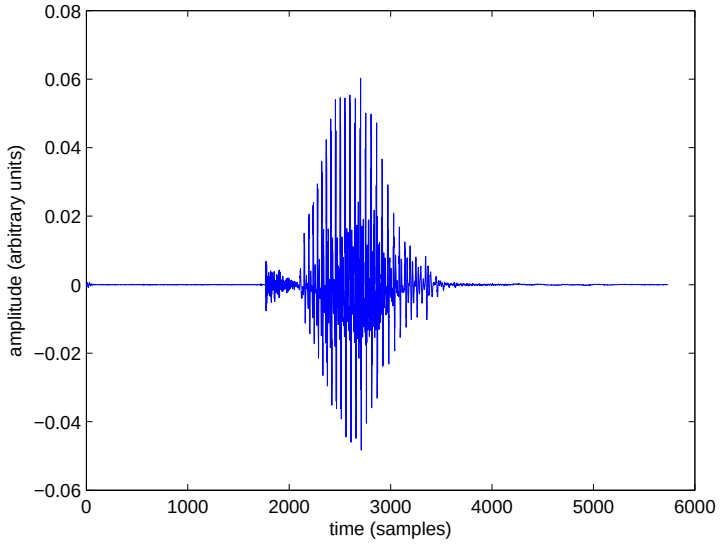

Figure 3.2: *The raw sound signal being processed.*

3. **Decimation**

   Next, the signal is decimated to lower the amount of data. To decimate the signal, it is low-pass filtered to maintain the Nyquist criterion and downsampled with a factor 100.

4. **Time Scaling**

   Because of the architecture of our HTM network, it can only analyze feature matrices, one or two dimensional, with a predefined size. To that end, we scaled all feature matrices to size 32×50: 32 filter outputs spanning the spectrum on 50 timepoints. Firstly, the first 7 time points are removed from the feature matrix, because they contain artefacts due to the settling time of the low-pass filter used in the previous step. Secondly, the remaining matrix is scaled to 32×50. A simple algorithm was used to realize this: When the utterance was shorter than 50 timepoints, extra feature vectors were added between the two neighbouring feature vectors that were closest (in a Euclidean sense) until the length of 50 was reached. This added vector is the mean of the neighbouring vectors. When the utterance was longer, neighbouring vectors that were closest were deleted until the length was 50 timepoints.

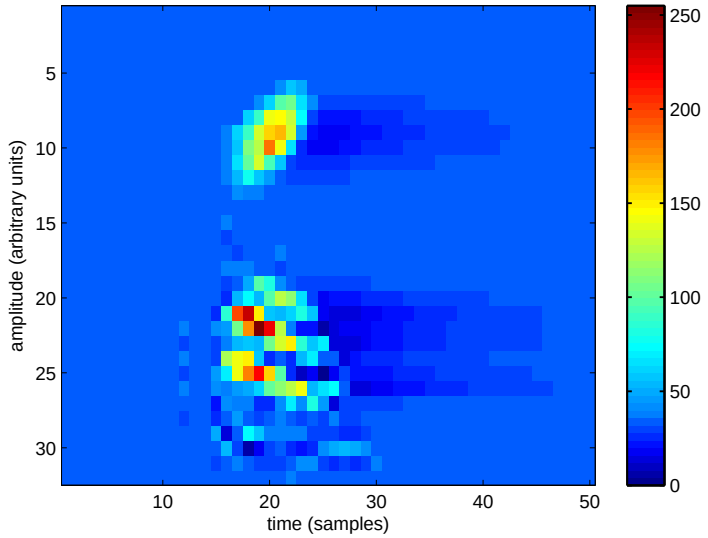

Figure 3.3: *Feature values at each sample point after all preprocessing stages. The feature numbers are depicted from high to low, mapping to high and low frequencies respectively.*

5. **Quantization**

   Finally, all values in the feature matrices are quantized with 8 bits to lower the amount of data, resulting in values between 0 and 255. This was done using Eq. 5.1.

$$x' = round\left\{\frac{x - min(x)}{max(x) - min(x)} \cdot (2^8 - 1)\right\} \tag{3.3}$$

   where *max* and *min* give the maximum and minimum value in the feature matrix and *round* rounds the number to the nearest integer. Note that it is important to quantize per utterance to normalize loudness differences between utterances. Furthermore, the minimum and maximum values must be taken from the whole feature matrix (and not for each filter), to maintain filter output differences.

   An example of the original raw sound signal and the outcome of preprocessing can be found in Fig. 3.2 and 3.3 respectively.

The obtained feature representations are converted to 8-bit BMP images. This was necessary due to the implementation of our HTM network. In the next chapter, we will elaborate on this

and other implementation issues.

Chapter 4

Implementation Issues

In this chapter we will present the implementation of our HTM for recognizing spoken digits. We use NuPIC to implement our HTM network. The learning and inference algorithms operating in each node are all written in C++, but the developer implements a network using Python. To implement such a network three steps have to be taken: creating the network architecture, training the network and testing the network. For our network, these steps are explained in detail below. We based most of our implementation on the NuPIC `Pictures` network, which is a HTM network for recognizing simple images. Consequently, we treated our auditory feature matrices as images.

All the Python code for our network can be found in Appendix A.

4.1 Creating the network architecture

The code listing for this step can be found in Appendix A.2.

Several issues have to be taken into consideration while designing the architecture of a HTM network.

- **Dimensionality of the sensor grid.**

  The grid of sensor nodes can be either one- or two-dimensional. Our first idea was to calculate auditory feature vectors for every sample point in each utterance and use a one-dimensional sensor grid to process them. Because of the state of the current algo-

rithms, this method could not give us satisfactory results. This is due to the fact that in the present version of NuPIC, a network is trained using time-varying data, but it cannot perform inference based on time-varying data (also called *time-based inference*). Time-based inference will be supported in a future release of NuPIC. Because of these difficulties, we used feature matrices instead of vectors. This lead to a two-dimensional sensor grid of nodes, one dimension being the feature values on a time point (32 in total), the other being time (50). We ended up with 32×50 feature matrices as explained in the previous chapter. This is of course a serious limitation because the data must be scaled in time to fit the network.

- **Number of nodes and their connectivity.**
  Our network consists of 2 levels of nodes. The sensor is a 32×50 grid receiving the values of our feature matrices. Level 1 contains a 8×10 grid of nodes, each node having a receptive field of 4×5 feature values. At level 2, one top node exists. This architecture is visualized in fig. 4.1b. The choice for this architecture was rather arbitrary, because the HTM learning and inference algorithms ensure robustness for every reasonable network structure. That is, there may be differences in performance using different network structures, but the principles remain the same. However, experimentation is needed to optimize this architecture.

4.2 Training the network

The code listing for this step can be found in Appendix A.3.

4.2.1 Training procedure

To speed up the training process, we only train one column of nodes at each level. At the bottom level, this column of 8 nodes spans 32×5 feature values. The sensor, implemented by the `ImageSensor` node, creates a window with size 32×5 and sweeps over the whole image from left to right[1]. The sensor window is sweeped over the images by iteratively moving this

[1]With that respect our implementation differs from the `Pictures` network, because it scans the image from left to right only, while the `Pictures` network scans the image in all directions. This is due the fact that we

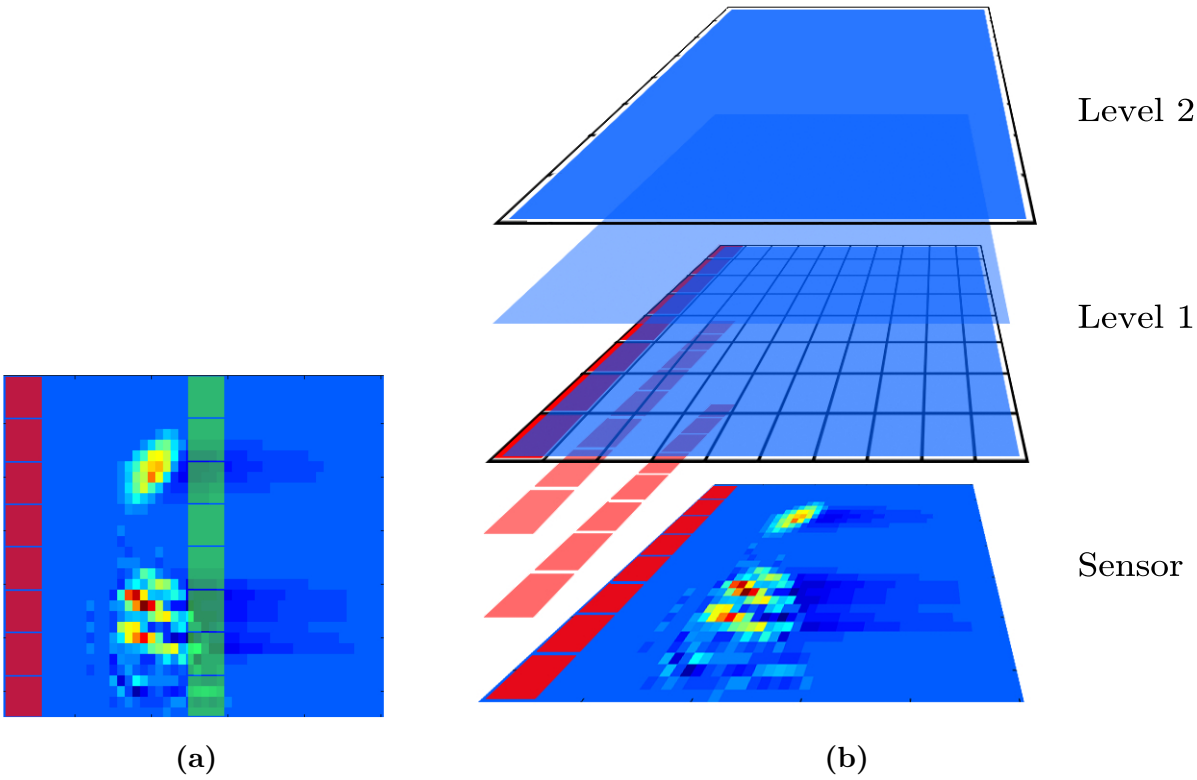

Figure 4.1: ***(a)*** *Colored rectangles represent the sensor window at different timepoints, the red rectangle at t=1 and the green rectangle at t=25. The feature values in the sensor window are the input to the column of first level nodes.* ***(b)*** *Our two-level architecture is visualized. At the the bottom (the sensor), the feature matrix is depicted. At the first level,* 8×10 *nodes are depicted. The top node is depicted above that, at the second level. The sensor feeds the column of nodes at the first level with feature values in the red rectangles representing the sensor window. The red rectangle corresponds to the one in* ***(a)****. After all images are scanned, the node states of nodes in the column are copied to the other nodes in their corresponding rows. The first level is now fully trained and set to inference mode. Next, the sensor window spans all nodes of the first level (the blue rectangle) and temporal groups of the first level nodes are projected to the top node at the second level.*

window one pixel (or feature value) to the right. The feature values in this window are the input to the column of eight nodes and the learning algorithms explained in Chapter 2 are performed on these values only. This process is visualized in Fig. 4.1a.

After all utterances, the node states (containing the states of the spatial and temporal poolers) are copied so that the other nine columns on the first level are now in the same state as the nodes in the trained column. The first level is now fully trained. Subsequently all nodes at this level are set to inference mode. The higher levels are trained in a similar way, but the

want the network only to look in the direction of time.

window size increases because one column of nodes on a higher level receives a larger part of the feature matrix. At the top node, the window spans the full image. This probably becomes more clear by looking at Fig. 4.1b.

In this figure, there are only two levels, so the window spans the whole image at the second level. In the next chapter, we will explore different architectures with more levels. We could, for example, have a three level network with a first level of 8×10 nodes, a second level of 4×5 nodes and one top node at the third level. In this example, the window at the first level spans 32×5 feature values, the window at the second level spans 32×10 feature values, and the window at the third level spans the whole 32×50 image.

4.2.2 Training parameters

Several parameters govern the working of the learning algorithms. Initially, we left most parameters set to the values found in the NuPIC `Pictures` network, except for maxDistance and sigma. We will discuss the effects of the parameters on the overall performance of the system in the next chapter. For maxDistance and sigma we obtained reasonable starting values in the following manner:

- maxDistance

  This parameter is only relevant in the Gaussian spatial pooler algorithm (in first level nodes). During learning, a new pattern is compared to existing coincidences. When the squared Euclidean distance between them is smaller than maxDistance, the new pattern is stored as a coincidence. When differences in feature values due to noise cause the network to pool them in different coincidences, the network will incorrectly infer that these feature values are due to different causes in the world. Therefore, to determine a good value for maxDistance an estimation of the level of noise in the data is needed. Every feature has a value between 0 and 255. One bottom level node has a receptive field of 4×5=20 feature values and we roughly estimate the noise in one feature value between 1 and 5, resulting in a value for maxDistance between $20^2 = 400$ and $(5 * 20)^2 = 10000$. By varying maxDistance between these values, the best (premature) results where ob-

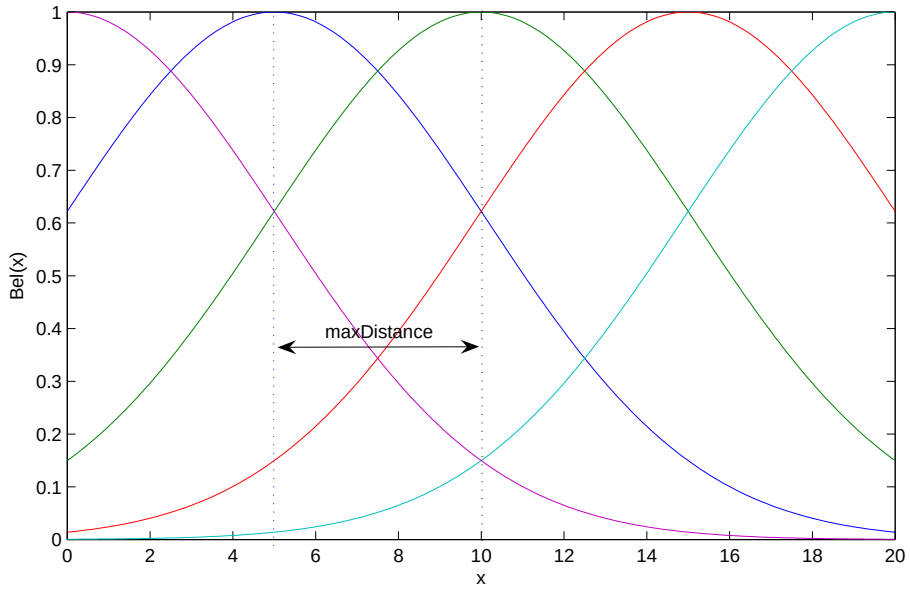

Figure 4.2: *Depicted are five belief distributions centered on previously seen coincidences {0,5,10,15,20}. They are plaxed* maxDistance *apart.*

tained with a value around 3600. In the next chapter, optimization of maxDistance is described.

- sigma

When a node is in inference state, the spatial pooler generates a belief vector over learned coincidences for a given input pattern. The belief in a coincidence is represented as a(n) (unnormalized) multi-dimensional Gaussian with the coincidence vector as its mean and a variance of sigma^2. These Gaussians are placed minimally maxDistance apart (see Fig. 4.2) and sigma determines on what belief value the distributions intersect. Two Gaussians with equal variance always intersect halfway between their means, so in the case of two Gaussians centered on 0 and maxDistance they intersect in $\frac{1}{2}$maxDistance. We can calculate the relation between maxDistance and sigma assumed that in $\frac{1}{2}$maxDistance the belief values given by the two Gaussians is a.

$$\exp\left\{-\frac{\frac{1}{2}maxDistance}{2 \cdot sigma^2}\right\} = a$$

$$maxDistance = 4 \ln\left(\frac{1}{a}\right) \cdot sigma^2$$

$$sigma = \frac{1}{\sqrt{4 \cdot \ln\left(\frac{1}{a}\right)}} \cdot \sqrt{maxDistance}$$

Because the belief value a is between 0 and 1 (exclusive), it follows that a reasonable value for *sigma* is in the same order of magnitude as $\sqrt{maxDistance}$. NUMENTA advises to start with *sigma* set to $\sqrt{maxDistance}$. In that case $\frac{1}{\sqrt{4 \cdot \ln\left(\frac{1}{a}\right)}} = 1$ and so $a \approx 0.78$. In our first experiments a larger *sigma* (around $2\sqrt{maxDistance}$, for which $a \approx 0.94$) gave us better results. Again, experimentation is needed to obtain an optimal value.

4.3 Testing the network

The code listing for this step can be found in Appendix A.4.

When the whole network is trained, all nodes are in inference state. Instead of sweeping over each feature matrix from left to right (as during the learning stage), the utterance as a whole is presented to the network. After all utterances in the test set are presented, we obtain the test results from a file showing the word category and the belief distribution over the word categories for each utterance. From this file, we can calculate the word error rate (WER) and other performance criteria.

We have described the implementation of our HTM network. Along with the code listings in Appendix A, a pretty good view of what issues have to be considered to implement an HTM network should be obtained. In the next chapter we will present the results we had with the network presented here, together with the results we obtained through tackling some apparent problems.

Chapter 5

Network Optimization

Several experiments were conducted with our HTM network to gain insight into the network, its inner workings and to optimize its performance. We will now present these experiments and their results. First, we will present the performance results of our initial network introduced in the previous chapter. Furthermore, we will optimize the most important parameters in this network, maxDistance and sigma.

Second, some apparent problems with this network and its input data are mentioned. We will provide several suggestions for solving these problems.

These suggestions form the basis of the rest of this chapter in which we will try to optimize both the auditory feature representations and the HTM network architecture and parameters.

5.1 A first HTM network for recognizing spoken digits

We start with the HTM network presented in the previous chapter. We varied across different parameter settings using this two-level architecture[1]. The following parameters were held constant:

- poolerAlgorithm: *Gaussian* in the spatial pooler of the nodes at the first level. *Product* in the spatial pooler of the nodes at the second level. These were the default values

[1] See Section 4.1 for more details about this architecture

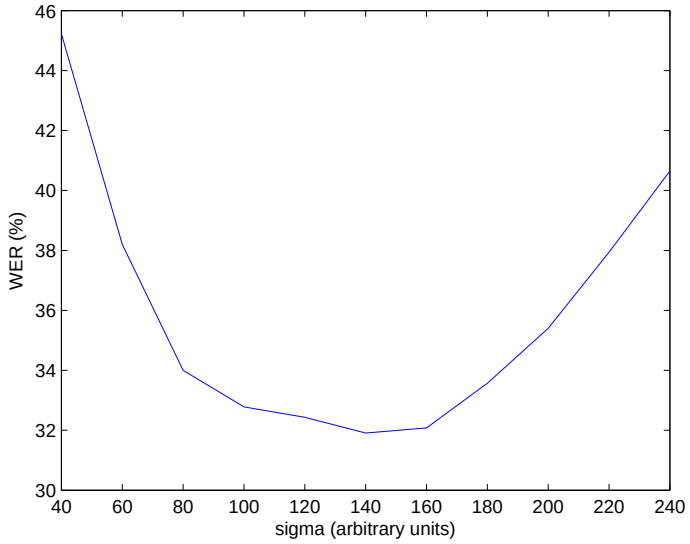

Figure 5.1: *Word error rate (WER) for various values of* SIGMA. *A* SIGMA *of 140 gave the lowest WER: 31.91%.*

in the NuPIC `Pictures` network, we found no obvious reasons to change them.

- SYMMETRICTIME: *False*. Time is not symmetric in our problem. The effect of this parameter was explained in Section 2.4.1.2.
- MAXGROUPSIZE: infinite. This was the default value in the NuPIC `Pictures` network and we found no reason to change it.
- GROUPERALGORITHM: sumProp. This was the default value in the NuPIC `Pictures` network, we found no reason to change it

We varied the following parameters:

- MAXDISTANCE
- SIGMA
- TRANSITIONMEMORY
- TOPNEIGHBOURS

These parameters only exist on the first level, because the second level is formed by a top node, which lacks these parameters. Initially, MAXDISTANCE and SIGMA were set to 3600 and 60 respectively, as explained in the previous chapter. We began with TRANSITIONMEMORY and TOPNEIGHBOURS set to 5 and 3, which are the values used in the NuPIC `Pictures` network.

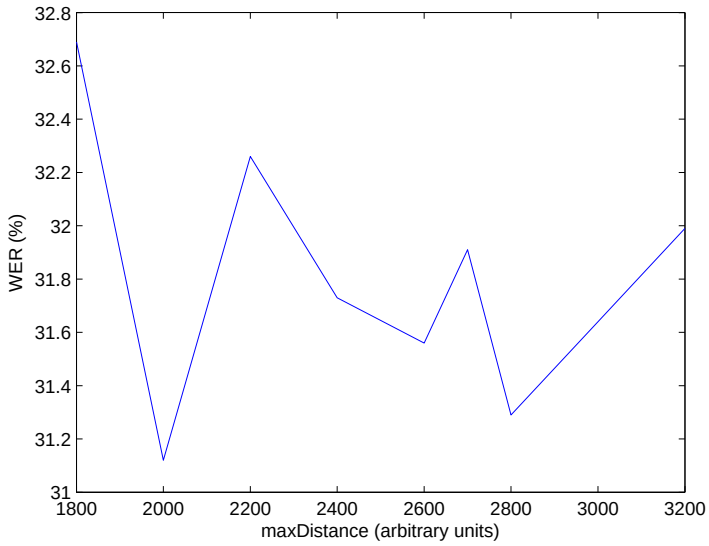

Figure 5.2: *WER for various values of* maxDistance. *A* maxDistance *of 2000 gave the lowest WER: 31.12%.*

First, we varied sigma, keeping all other parameters constant. The word error rate (WER) for various values of sigma is depicted in Fig. 5.1. As can be seen, a sigma value of 140 gave us the best results.

To optimize maxDistance, we had to change sigma too, because they depend heavily on each other. We used the relation between the current maxDistance (3600) and the optimal value found for sigma (140) to calculate sigma for other values of maxDistance. This relation is $sigma \approx 2.33\sqrt{maxDistance}$[2]. The results of this variation (other things being equal) are shown in Fig. 5.2. These results do not show a clear pattern, but the changes in WER are minimal. Some tweaking of sigma (to 150) gave our best result with this architecture: a WER of 30.77%. Varying transitionMemory and topNeighbours did not affect the WER significantly, and only for the worse.

The number of coincidences and temporal groups learned at the first level are the same for each row, because the states of 8 nodes (in a column) are copied to their respective rows[3]. The number of coincidences and temporal groups learned at the first level of the network with

[2]See Section 4.2 for more details about the relation between sigma and maxDistance
[3]See Section 4.2 for more details about the training procedure

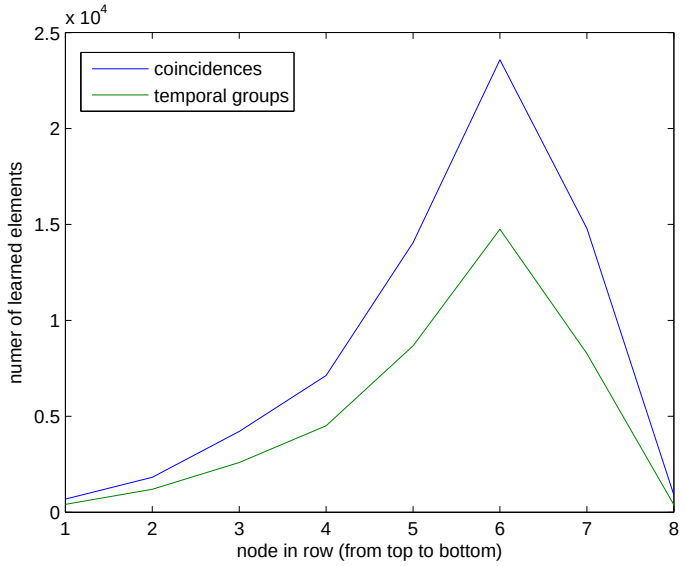

Figure 5.3: *The network with parameter setting* {MAXDISTANCE=*2000,* SIGMA=*150,* TRANSITIONMEMORY=*5,*TOPNEIGHBOURS=*1*} *learned the following numbers of coincidences and temporal groups in the training column of nodes at level 1.*

our best performing parameter setting {MAXDISTANCE=2000, SIGMA=150, TRANSITIONMEMORY=5,TOPNEIGHBOURS=1} can be found in Fig. 5.3. The maximum number of coincidences in a single node when MAXDISTANCE is 0 is $50 \times 2420 = 121000$[4]. The ratio between learned coincidences and the maximum possible number of coincidences is approximately 0.19. In other words, roughly 5 5×4 feature value grids are pooled with the same coincidence. On average, the ratio between the number of temporal groups and the number of coincidences is 0.59.

It's quite difficult to interpret these numbers, but at first sight the number of input patterns pooled per coincidence seems quite low. A possible way of decreasing the number of coincidences is to increase MAXDISTANCE. For example, when we set MAXDISTANCE to 14400, the number of coincidences drops with a factor 12.68 and the number of temporal groups with a factor 11.52 (See Fig. 5.4). The WER of this network is 37.69%, which is a significant decrease in performance, but not as drastically as one might expect.

The confusion matrix for our best performing parameter setting looks as follows:

[4]Because every training pattern in the train set (2420) is scanned in 50 iterations, every node receives $50 \times 2420 = 121000$ input patterns (4×4 patches of feature values) during the training

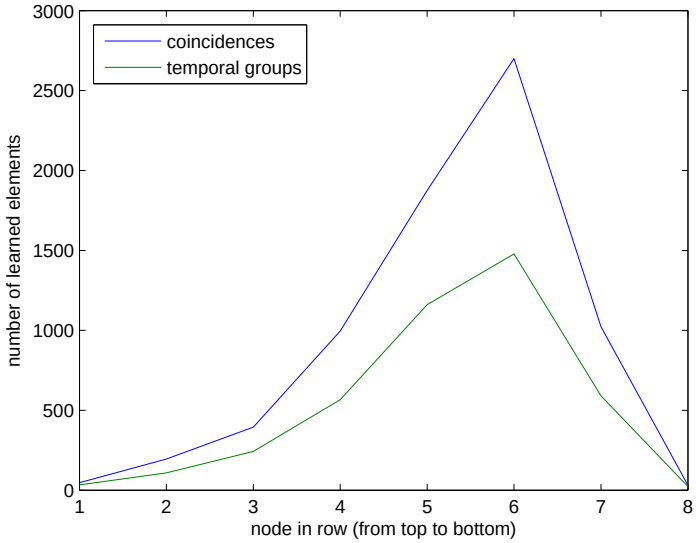

Figure 5.4: *The network with parameter setting* {maxDistance=*14400,* sigma=*276,* transitionMemory=*5,*topNeighbours=*1*} *learned the following numbers of coincidences and temporal groups in the training column of nodes at level 1.*

	oh	*one*	*two*	*three*	*four*	*five*	*six*	*seven*	*eight*	*nine*	*zero*
oh	**0.76**	0.03	0.00	0.01	0.00	0.01	0.01	0.04	0.08	0.05	0.02
one	0.01	**0.56**	0.00	0.02	0.01	0.03	0.07	0.08	0.15	0.01	0.07
two	0.00	0.00	**0.51**	0.25	0.07	0.08	0.00	0.09	0.00	0.01	0.00
three	0.00	0.00	0.02	**0.80**	0.04	0.05	0.00	0.09	0.01	0.00	0.00
four	0.02	0.00	0.05	0.34	**0.51**	0.02	0.00	0.04	0.00	0.03	0.00
five	0.05	0.00	0.03	0.14	0.04	**0.59**	0.00	0.12	0.03	0.01	0.00
six	0.00	0.05	0.00	0.01	0.00	0.00	**0.88**	0.01	0.04	0.00	0.01
seven	0.00	0.01	0.01	0.01	0.01	0.01	0.00	**0.92**	0.03	0.00	0.00
eight	0.00	0.03	0.00	0.02	0.00	0.00	0.02	0.12	**0.80**	0.00	0.01
nine	0.09	0.00	0.03	0.09	0.04	0.00	0.00	0.18	0.04	**0.54**	0.00
zero	0.02	0.05	0.00	0.01	0.00	0.00	0.12	0.01	0.03	0.01	**0.75**

where element (i, j) respresents the fraction of the number of times i is classified as j. Some confusions deteriorate the performance of our system significantly, Most notably are 'four' as

'three' and 'two' as 'three'. Visually, we could observe the similarities between these confused digits in our feature matrices (which are represented as images). On the other hand, the confusions one would expect (phonetically) between for example 'oh'-'zero' and 'nine'-'five' occur much less often.

5.2 Apparent problems and possible solutions

The results in the previous section show that some learning has occured, but also that there may be some problems that reduce the performance of the network significantly. We will now present some apparent problems and possible solutions.

5.2.1 The problem of truncation of input data

Through close inspection of the data, some utterances seem to be truncated either in the beginning or the end or both. Therefore these utterances may not be correctly learned or recognized. A simple solution is to increase the size of the feature matrices in the time dimension to 64, obtaining feature matrices of size 64×32. This was done by rescaling the feature matrix to 32×64 instead of 32×50 during preprocessing.

We will discuss the effect of this change in Section 5.3.1.

5.2.2 The problem of modelling silence

Due to Meddis' Inner Hair Cell model and quantization during preprocessing (both discussed in Section 3.2), silence is represented by different values across utterances. In the hair cell model, some period of stimulation lowers the probability of neurotransmitter release after that stimulation period. So, in utterances with relatively longer periods of stimulation, relatively lower values (representing lower probabilities) exist after the period of stimulation. These lower probabilities are even lower than the probabilities during a period of silence and therefore the smallest value in the feature representation differs across utterances. Due to our quantization

formula

$$x' = round\left\{\frac{x - min(x)}{max(x) - min(x)} \cdot (2^8 - 1)\right\} \tag{5.1}$$

the smallest value $min(x)$ is represented as 0, while silence has higher values depending on the value of $min(x)$ and $max(x)$.

Because silence is represented differently across utterances, the netwerk may have problems discriminating silence from non-silence. Therefore a different feature representation may be more appropriate. An alternative is discarding the Meddis' Inner Hair Cell model and average energy over an exponential time window after it has gone through the gammatone filterbank. By definition, silence is then represented with 0 because there is no energy in the signal there.

We will examine the effect of this change in the feature representation in Section 5.3.2.

5.2.3 The problem of too much information in the input data

The number of possible values for a certain feature value is quite large (255). This could make it difficult for the network to discriminate between noise and important causes in the world. Therefore it may be fruitful to lower the amount of possible values for a feature value. In our implementation the feature matrices are converted to 8-bit BMP images, so the feature values must still lie between 0 and 255. A possible way of lowering the amount of possible values to n may be to change our quantization formula to:

$$x' = round\left\{\frac{2^8 - 1}{n - 1} \cdot round\left\{\frac{x - min(x)}{max(x) - min(x)} \cdot (n - 1)\right\}\right\} \tag{5.2}$$

which assigns to each original value one of the n quantized values which still lie between 0 and 255.

In Section 5.3.3 we will examine whether lowering the amount of possible values helps the network to discriminate between the digits.

5.2.4 The problem of first-order temporal learning

Temporal groups are formed using a first-order algorithm, which is discussed in Section 2.4.1.2. A significant amount of temporal structure is therefore lost in temporal groups. For example, assume that a "rising" sequence $100 \rightarrow 120 \rightarrow 130$ and a "falling" sequence $130 \rightarrow 120 \rightarrow 100$ occur frequently at the input of a node. Of course we would like to have separate representations for these two different sequences, but due to the first-order algorithm they are represented by the same temporal group.

A solution to this problem is not to use the feature values as input data, but the difference between feature values from 2 consecutive samples. The sequences in the previous example then become (assuming that they both occur at the beginning of an utterance) $0 \rightarrow 20 \rightarrow 10$ and $0 \rightarrow -20 \rightarrow -10$, which can be discriminated by the temporal pooler.

In Section 5.3.4 the effect of using this delta feature representation is discussed.

5.3 Auditory feature respresentation optimization

The effects of the changes in the auditory feature representation proposed in the previous section are now examined. Due to practical time constraints, we could not examine the independent effects of these changes. We took an incremental approach instead, changing the feature representation in the following order (when the change improved the performance):

1. enlarging feature matrices
2. spectral energy feature representation
3. lowering amount of possible values
4. delta feature representation

We used a two-level network with a first level of 16×8 nodes to test the effects of these changes. The parameters besides maxDistance and sigma are the same as in our initial network.

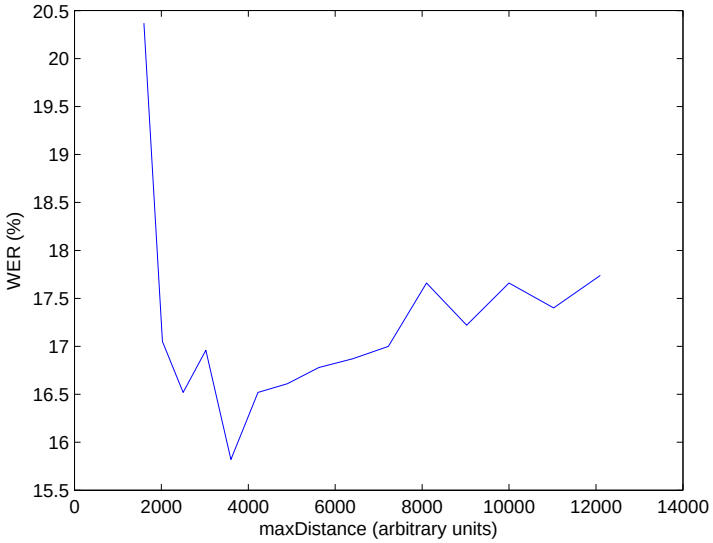

Figure 5.5: *The network with parameter setting* {maxDistance=*2000,* sigma=*150,* transitionMemory=*5,*topNeighbours=*1*} *learned the following numbers of coincidences and temporal groups in the training column of nodes at level 1.*

5.3.1 The effect of enlarging feature matrices

To use the larger feature matrices, the size of the first level of the network must be changed from 8×10 nodes to, for example, 8×16. We will use this number of nodes on the first level to optimize our auditory feature representations, but of course other node grid configurations are also possible, i.e. 4×8 and 16×32.

We will discuss architecture and parameter optimization in Section 5.4.

We trained and tested our network with the larger feature representations, using our best performing network parameters (corrected for the change in size of the level one node grid). The WER decreased slightly, from 30.77% to 30%. This is not a significant decrease, but we will still use the larger feature matrices because of the convenience it gives us to experiment with other node grid sizes.

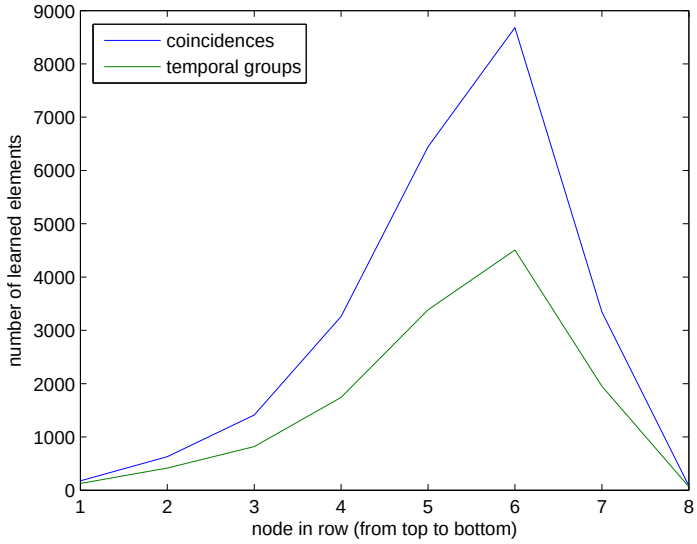

Figure 5.6: *The network trained and tested with spectral energy feature representations and with parameter setting* {MAXDISTANCE=*3600,* SIGMA=*138,* TRANSITIONMEMORY=*5,*TOPNEIGHBOURS=*1*} *learned the following numbers of coincidences and temporal groups in the training column of nodes at level 1.*

5.3.2 The effect of using a spectral energy feature representation

In Fig. 5.5 various settings of MAXDISTANCE are plotted against the WER of the network trained and tested with spectral energy feature representations. The value of SIGMA is obtained by using the relation $sigma \approx 2.33\sqrt{maxDistance}$ again. The network with MAXDISTANCE=3600 and SIGMA=138 (no value of SIGMA performed better) had the lowest WER, namely 15.82%. The number of coincidences and temporal groups of that network are depicted in Fig. 5.6. The ratio between the number of coincidences and temporal groups is the same as the one found in the initial network.

The significant increase in performance in terms of WER is probably related to more constistently modelling silence across utterances as explained in Section 5.2.1.

5.3.3 The effect of lowering the amount of possible values

We lowered the amount of possbible values from 255 (8 bit) to 4 (2 bit), 8 (3 bit) and 16 (4 bit) in that respective order. The quantization formula in Section 5.2.2. was used to achieve

this.

The network was trained using different values of maxDistance and sigma. These depend on the number of possible values. For example, using the 2 bit representations, there are 4 possible values between 0 and 255: 0, 85, 170 and 255. As a consequence, values for maxDistance between 0 and $85^2 = 7225$ are equivalent (and values between 7225 and $170^2 = 28900$ etc. too). So, we used only these settings for maxDistance and calculated sigma using $sigma \approx 2.33\sqrt{maxDistance}$.

The performance of the network deteriorated using 2 bit spectral energy feature representations, the WER increased from 15.82% to around 20%. This was roughly the same for all settings of maxDistance and sigma. With 3 bit representations, the lowest WER we obtained was 12.85%. This was achieved using maxDistance=5184 and sigma=168. With 4 bit representations, the network performed slightly better. Using maxDistance=4624 and sigma=159, the WER was 12.33%. We also trained and tested the network with 5 bit representations, but the WER deteriorated using the representations: equivalent settings for maxDistance and sigma (as used with the 4 bit representations) increased the WER with 2-3%.

The network performs best when trained and tested with 4 bit spectral energy feature representations. Probably this number of values is a good balance between discriminative and generalization power.

5.3.4 The effect of using a delta feature representation

We calculated 4 bit spectral energy delta representations in the following way. First, after the signal was decimated and the first 7 frames were removed (due to to the settling time of the low-pass filter used at decimation), linear regressions through 5 consecutive time points where calculated (for each filter). The slope of the regression is the delta value of the middle time point, so the regression is calculated from -2 time points to +2 time points relative to the base time point. If we would quantize these values using the minimum and maximum slopes, silent parts will have different values across utterances. During silent parts the delta values are of course 0, but if the maximal and minimal slope values are not symmetric relative to zero (x

and $-x$ respectively) the quantization gives different values for 0 across utterances. A solution is to quantize with $-maximum$ and $maximum$ when $maximum$ is larger than $-minimum$, or quantize with $minimum$ and $-minimum$ when $-minimum$ is larger than $maximum$.

The performance of the network did not increase using delta representations. The lowest WER we found was 19.93%, with maxDistance=1156 and SIGMA=79.

This result indicates that the problem described in Section 5.2.4. does not have a significant negative influence on the performance of the network. To test this hypothesis, we trained the network with the 4 bit spectral energy representations and with MAXDISTANCE=4624 and SIGMA=159. Furthermore, we set MAXGROUPSIZE=1. This forces the network to create temporal groups with only one coincidence. As a consequence, the network does not use temporal information between coincidences in the training and testing algorithms[5]. The performance of the network increased as a result of this change, the WER decreased to 11.71%. This shows the network does not use temporal information obtained through temporal groups.

We have obtained closer to optimal input representations, namely 4 bit spectral energy feature representations. A network trained with the new representations, MAXDISTANCE=4624, SIGMA=159 and MAXGROUPSIZE=1 results in the following confusion matrix:

[5]However, short term temporal information is still present within the coincidences.

	oh	*one*	*two*	*three*	*four*	*five*	*six*	*seven*	*eight*	*nine*	*zero*
oh	**0.88**	0.01	0.00	0.01	0.00	0.04	0.02	0.00	0.00	0.01	0.01
one	0.02	**0.88**	0.00	0.02	0.00	0.03	0.00	0.00	0.00	0.01	0.03
two	0.00	0.01	**0.85**	0.03	0.03	0.05	0.00	0.04	0.00	0.00	0.01
three	0.00	0.00	0.01	**0.88**	0.10	0.01	0.00	0.01	0.00	0.00	0.00
four	0.00	0.00	0.00	0.21	**0.77**	0.01	0.00	0.00	0.00	0.00	0.01
five	0.04	0.01	0.01	0.01	0.01	**0.89**	0.00	0.00	0.01	0.00	0.01
six	0.01	0.04	0.00	0.01	0.00	0.00	**0.92**	0.00	0.00	0.00	0.02
seven	0.00	0.00	0.00	0.01	0.01	0.00	0.00	**0.98**	0.00	0.00	0.00
eight	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.05	**0.92**	0.01	0.01
nine	0.01	0.00	0.00	0.05	0.02	0.00	0.00	0.06	0.01	**0.88**	0.00
zero	0.01	0.07	0.01	0.01	0.00	0.00	0.02	0.00	0.01	0.00	**0.88**

There are less severe confusions in this network than in our initial network. However, in many cases 'four' is still being confused with 'three'(21%) and vice versa (10%).

The code listings for this network differ only in the code that creates and trains the network. These are added in Appendices A.5 and A.6 respectively.

5.4 Architecture en parameter optimization

Now that we have obtained a closer to optimal feature representation, we can optimize the network architecture and parameters. We take a two-step approach in optimizing our architecture:

1. Modify the two-level architecture by changing the number of nodes at the first level
2. Add another or more levels to the best performing two-level architecture

At each of these steps the parameters for that architecture are optimized.

5.4.1 Effects of changing the number of nodes at the first level

We tested 2 other possible configurations of first level nodes besides 16×8, namely: 4×8 and 16×32.

A network with 4×8 nodes at the first level gave slightly worse results than a network with 8×16 nodes when trained with the same values for maxDistance and sigma. The lowest WER we found was 12.50% with maxDistance=7225 and sigma=198.

At first sight this may seem quite strange, because in the 4×8 network coincidences are 8×8 patches of feature values, whereas in a network with 16×8 nodes coincidences are 4×4 patches of feature values. Hypothesising that the average distance between two feature values still belonging to the same cause was $\frac{\sqrt{4624}}{4 \times 4} = 4.25$ (see [6]) in the 8×16 network, we trained a network with 4×8 nodes at the first level with $(8 \times 8 \cdot 4.25)^2 = 73984$. This deteriorated the performance significantly, namely a WER of 35.78%.

A network with 16×32 nodes at the first level gave slightly better results than a network with 16×8 nodes when trained with the same values for maxDistance and sigma. The lowest WER we found was 11.63% with maxDistance=4624 and sigma=159. Again we found deterioration in performance when we changed maxDistance with respect to the size of patches of feature values. Using maxDistance= $(2 \times 2 \cdot 4.25)^2 = 289$, the performance deteriorated drastically, a WER of 59.79%.

The deteriorations we found after changing maxDistance with respect to the number of nodes and the stability in performance found when we used values for maxDistance rather close to the optimal values we found for the 8×16 network, indicate that the squared Euclidean distance between input pattterns with similiar causes is the same across these different networks. This is plausible, because for example a difference of less than 4 between two 8×8 patches of feature values is in the same way due to noise as is a difference of less than 4 between two 2×2 patches of feature values. When we trained and tested the networks using smaller values than the *level of noise* found through experimentation, so maxDistance< 4624, the performance drops drastically, i.e. a WER of 59.79% with maxDistance$= 289$ in the 16×32 network. On the other hand, when we estimate maxDistance too high, so when too many patterns with

[6] This is the square root of the optimal value of maxDistance in the 16×8 network, divided by the number of feature values projecting to one first level node in the same network.

different causes are pooled together, the performance also drops, but less severe, i.e. a WER of 35.78% with maxDistance= 73984 in the 4×8 network. The network is better able to discriminate the digits when using a coincidence with different (but relatively similiar) causes, than when using different coincidences with the same underlying cause. Another advantage of estimating maxDistance too high instead of too low is shorter training and inference time. This is a trend we saw in all of our experiments.

We varied the parameters topNeighbours, transitionMemory, maxGroupSize, poolerAlgorithm and grouperAlgorithm of the nodes at the first and top level. None of these other parameter settings lead to improvement in performance.

5.4.2 Effects of adding more levels

So far the HTM networks we tested had only two levels. We can add another level of nodes to our best performing network (16×32 nodes at the first level, maxDistance=4624, sigma=159 and maxGroupSize=1). We tested two networks with three levels: a network with a second level of 8×16 nodes and one with a second level of 4×8 nodes. The first level node parameters remained constant, the second level node parameters are the same except for using the Dot spatial pooler algorithm.

Apart from a severe increase in training time[7], the WER did not decrease in both of the three level networks, but remained 11.63%. This shows that a more complex hierarchy of nodes is not necessary for this problem.

We varied the parameters topNeighbours, transitionMemory, maxGroupSize, poolerAlgorithm and grouperAlgorithm at the second level. None of these other parameter settings lead to an improvement in performance. However, when we set maxGroupSize to infinity again the WER of the network is 11.71%. As opposed to the two level network, the performance did not decrease due to the temporal information in temporal groups.

[7]The training time increased from roughly 1 hour for the two level network to 48 hours for the three level network. This could be due to an increase in memory usage.

Chapter 6

Conclusion

In this study we have investigated the possible benefits of Hierarchical Temporal Memory networks to automatic speech recognition. To that end, we implemented an HTM network through the use of existing software tools with the goal of recognizing the 11 words in the TIDIGITS database.

After optimizing different aspects of the input patterns and the network itself, our network has a word error rate of 11.63%. This result can be considered reasonably well, because the train set was considerably small (comparison with HMM?) and the system is speaker independent.

The main conclusions gained in this research are threefold:

1. We used an existing software package for the implementation of our network. The software as well as the HTM framework is still actively in devlopment. In theory, HTM network are capable of recognizing patterns that are inherently time-based, but this feature was not yet implemented in our software tools.
   Because speech is time-based, we needed to create a workaround for this problem. We only considered isolated digits, so we could choose a time window that would fit all utterances in the train set. The training of the network was done using multiple time slices of the utterance, but in the inference stage utterances were presented to the network as a whole. Because of this choice speech was modelled speech in two spatial dimensions: frequency and time.
   Probably this is not an optimal solution, and when time-based inference in HTM net-

works is present, these two dimensions are unnecessary. We could then train the network with auditory features calculated from one time frame and also present such feature vectors during inference. HTM networks could, in theory, make predictions about future input (on every level in the hierarchy) and calculate beliefs over every category at each time frame.

2. The term *"Hierarchical Temporal Memory"* indicates that **Hierarchy** and **Time** are important elements in this technology. For our particular problem however, the importance of these intuitively elegant elements or the way in which they are implemented is disputable.

   Hierarchy is found in the hierarchical topology of HTM networks. When we added intermediate levels to our best performing two-level network, the performance did not increase. This could indicate that a more complex hierarchy is not necessary for solving our problem.

   HTM networks form temporal groups of patterns that occur close in time frequently. When we force our network to create singleton temporal groups, we essentially switch off the temporal component of HTM networks. In all our networks, the performance increases or remains constant after this adaptation, indicating that modelling of time through temporal groups is not helping the network in recognizing the utterances. The problem could also lie in the way these temporal groups are represented[1] or computed.

3. The recognition that takes place in our network is presumably owed to the spatial pooling of input patterns and the representation of these pools or coincidences during inference as Gaussians distributions. The network learns to associate certain spatial configurations[2] of coincidences with certain categories.

It would be interesting to see how HTM networks with time-based inference perform. Furthermore, it would be fruitful to investigate the different aspects of the current HTM learning and inference algorithms more closely, especially the spatial and temporal pooling algorithms. Examples of possible adaptations are: dynamically changing coincidences (instead of static

[1] As exclusive sets of coincidences, without any internal structure
[2] Time on the horizontal axis, frequency on the vertical axis

ones), non-exclusive temporal grouping [3] and temporal groups with more internal temporal structure.

[3] So that coincidences can be part of multiple temporal groups

Appendix A

Python Code

The Python code for implementing our HTM network is separated in four listings.

- **RunOnce.py**

  This listing contains a wrapper function which subsequently creates, trains and test the network using the listings below.

- **CreateNetwork.py**

  This listing contains a function which creates the network architecture and saves it to a file.

- **TrainNetwork.py**

  This listing contains a function which trains the network and saves the trained network to a file.

- **RunInference.py**

  This listing contains a function which test the network and outputs the results to a file.

A.1 RunOnce.py

```
#!  /usr/local/bin/python

from CreateNetwork import createNetwork
from TrainNetwork import trainNetwork
from RunInference import runInference
from nupic.analysis import Visualizer

#============================================================================
# The main routine
#============================================================================
def runOnce():

    # Create the network
    createNetwork(untrainedNetwork)

    # Train the network
    trainNetwork(untrainedNetwork, trainedNetwork)

    # Test the network
    runInference(trainedNetwork, testResults)

    print "Speech run complete.  Results are in 'report.txt'"

#============================================================================
# List of parameters used in the example
#============================================================================

# Various file names
untrainedNetwork   = "untrained_speech.xml"    # Name of the untrained network
```

```
trainedNetwork      = "trained_speech.xml"      # Name of the trained network
testResults         = "test_results.txt"         # File containing inference
results for each test pattern



#================================================================================
# When invoked from command line, create network, train it, and run inference
#================================================================================
if __name__== '__main__':
  runOnce()
```

A.2 CreateNetwork.py for the initial HTM network

```
#!  /usr/local/bin/python

from nupic.network import Network, Node, Region
from nupic.pynodes import *
from nupic.pynodes.ImageSensor import *
from nupic.pynodes.PyNode import *
from nupic.network import SimpleFanIn, SimpleSensorLink, CreateNode, SingleLink
from nupic.network.nodetools import \
  VectorFileSensorTool, VectorFileEffectorTool, \
  Zeta1NodeTool, Zeta1TopNodeTool, GetNTAPlugins

def createNetwork(untrainedNetwork):

  # Create empty network
  net = Network()

  # Set image width and height
  imageWidth = 50
  imageHeight = 32

  # Create sensor node
  sensorNode = CreateNode("nupic.pynodes.ImageSensor.ImageSensor",
    phase=0, image=(imageWidth*imageHeight) , category=1,
    imageWidth=imageWidth, imageHeight=imageHeight)

  # Add sensor to the network
  net.addElement("sensor",sensorNode)

  # Create first level template node
```

```
templateNodeLevel1 = Zeta1NodeTool(phase=1,
  poolerAlgorithm='gaussian',
  symmetricTime=False,
  transitionMemory=5,
  topNeighbors=3,
  maxGroupSize=1000000,
  grouperAlgorithm='sumProp',
  outputElementCount=200000,
  maxDistance=3600,
  detectBlanks=1)

# Create first level of nodes
myRegion = Region([8,10], templateNodeLevel1)

# Add first level to network
net.addElement("Level1", myRegion)

# Link sensor to first level
net.link("sensor", "Level1", SimpleSensorLink([imageHeight,imageWidth],"image",
"bottomUpIn"))

# Create second level top node
topnode = CreateNode("Zeta1TopNode",
  phase=2,
  categoriesOut=11,
  mapperAlgorithm='sumProp',)

# Add top node
net.addElement("Level2", topnode)

# Link first level to top level
```

```
  net.link("Level1", "Level2", SimpleFanIn())

  # Link sensor to top level (for sending the category of the training pattern)
  net.link("sensor", "Level2", SingleLink("category", "categoryIn"))

  # Create effector for output to a file
  net.addElement("FileOutput", VectorFileEffectorTool(phase=3))

  # Link category outputs and top node output to the file writing effector
  net.link("Level2", "categoriesOut", "FileOutput", "dataIn")
  net.link("sensor","category","FileOutput","dataIn")

  # save the network in a file
  net.writeXML(GetNTAPlugins(), untrainedNetwork)
  print "Network created."

#===============================================================================
# If invokedfrom the command line, just create network and save it
#===============================================================================
if __name__== '__main__':
    createNetwork('speech.xml')
    print "Saved HTM network to file 'speech.xml'"
```

A.3 TrainNetwork.py for the initial HTM network

```
#!  /usr/local/bin/python

from nupic.session import Session
import sys
import os

def trainNetwork(untrainedNetwork, trainedNetwork):

  # Prepare session
  session = Session(os.path.join("sessions", "speech"))
  session.start()

  # Load network file and set parameters
  session.loadNetwork(untrainedNetwork)
  print 'Network loaded.'

  # Set sigma
  session.execute('Level1.*', ['setParameter', 'sigma', '60'])

  # Set type of sweeps the sensor must make (only to the right)
  session.execute("sensor", "self.sweepTypes=['right']")

  # Directory containing 11 directories.  Each of these 11 directories contains
  all train patterns of a certain digit
  trainDirectory =
  ""

  # Load BMP images
  session.evalPyNodeExpr("sensor", "self.loadMultipleImages('"+ trainDirectory
```

```
+
  "', 'bmp')")

  #--------------------------------------------------------
  # Train Level 1
  #--------------------------------------------------------

  print 'Training level 1...'

  # Set the width and height of the sensor window
  session.execute('sensor', 'self.enabledWidth = 5')
  session.execute('sensor', 'self.enabledHeight = 32')

  # Reset sensor
  session.evalPyNodeExpr('sensor', 'self.reset()')

  # Disable all nodes
  session.disableNodes()

  # Compute number of iterations
  iterations = session.evalPyNodeExpr('sensor','self.numIterations()')

  print 'Number of iterations:', iterations, '.'

  # Enable sensor
  session.enableNodes("sensor")

  # Enable one column of nodes
  session.enableNodes("Level1\[.*,0]")

  # Set column of nodes to learning mode
```

```
session.execute('Level1\[.*,0]', ['setLearning', '1'])

# Train the column of nodes on the train patterns
session.compute(iterations)

# Unset column of nodes from learning mode
session.execute('Level1\[.*,0]', ['setLearning', '0'])

# Clone every node state in the column to the nodes in its row
session.sendRequest('nodeCopyState Level1\[0,0] Level1\[0.*]');
session.sendRequest('nodeCopyState Level1\[1,0] Level1\[1.*]');
session.sendRequest('nodeCopyState Level1\[2,0] Level1\[2.*]');
session.sendRequest('nodeCopyState Level1\[3,0] Level1\[3.*]');
session.sendRequest('nodeCopyState Level1\[4,0] Level1\[4.*]');
session.sendRequest('nodeCopyState Level1\[5,0] Level1\[5.*]');
session.sendRequest('nodeCopyState Level1\[6,0] Level1\[6.*]');
session.sendRequest('nodeCopyState Level1\[7,0] Level1\[7.*]');

# Enable all nodes on the first level
session.enableNodes("Level1.*")

# Set column of nodes to inference mode
session.execute('Level1.*', ['setInference', '1'])

#-------------------------------------------------------
# Train Level 2
#-------------------------------------------------------

print 'Training level 2...'

# Set the width and height of the sensor window
```

```
session.execute('sensor', 'self.enabledWidth = 50')
session.execute('sensor', 'self.enabledHeight = 32')

# Reset sensor
session.evalPyNodeExpr('sensor', 'self.reset()')

# Compute number of iterations
iterations = session.evalPyNodeExpr('sensor','self.numIterations()')

print 'Number of iterations:', iterations, '.'

# Enable top node
session.enableNodes("Level2.*")

# Enable one column of nodes
session.execute('Level2.*', ['setLearning', '1'])

# Train the top node on the train patterns
session.compute(iterations)

# Unset top node from learning mode
session.execute('Level2.*', ['setLearning', '0'])

# Set top node to inference mode
session.execute('Level2.*', ['setInference', '1'])

# Enable all nodes
session.enableNodes()

#-----------------------------------------------------------------
# Save the network and retrieve trained network file from bundle
```

```
  #-----------------------------------------------------------------

  print 'Training complete.'

  # Save the trained network
  session.saveRemoteNetwork(trainedNetwork)

  # Stop session
  session.stop()

  # Retrieve the network file
  session.getLocalBundleFiles(trainedNetwork)

  # Clean up unused files
  session.setCleanupLocal(True)

#============================================================================
# If invoked from the command line, do nothing
#============================================================================
```

A.4 RunInference.py

```
#!  /usr/local/bin/python

from nupic.session import Session
import sys
import os

#============================================================================
# The main routine
#============================================================================
def runInference(trainedNetwork, resultsFile):

  # Prepare session and data files we will use
  session = Session(os.path.join("sessions", "speech"))
  session.start()

  # Load network file.  All nodes will be enabled by default
  session.loadNetwork(trainedNetwork)

  # Set the output file
  session.execute("FileOutput", ("setFile", resultsFile))

  print 'Running inference...'
  session.execute("FileOutput",
      ("echo", "Numbers show distributions over groups"))

  # Set the image sensor to flash mode, so that images are presented as a whole
  session.evalPyNodeExpr("sensor", "self.prepareForFlash()")

  # Directory contains all test patterns
```

```
  testDirectory =
  ""

  # Load BMP images
  session.evalPyNodeExpr("sensor", "self.loadMultipleImages('"+ testDirectory +
  "', 'bmp')")

  # Reset the sensor
  session.evalPyNodeExpr('sensor', 'self.reset()')

  # Compute number of iterations
  iterations = session.evalPyNodeExpr('sensor','self.numIterations()')

  # Compute belief distributions over categories for each test image
  session.compute(iterations)

  # Stop session
  session.stop()

  # Save file with results
  session.getLocalBundleFiles(resultsFile)

  # Clean up unused files
  session.setCleanupLocal(True)

  print "Inference complete; results in ", resultsFile

print 'Starting session...'

#===============================================================================
# When invoked from command line, run inference
```

```
#============================================================================
if __name__== '__main__':
  print 'Starting session...'
  runInference(trainedNetwork, trainingResults)
```

A.5 CreateNetwork.py for the optimized HTM network

```
#!  /usr/local/bin/python

from nupic.network import Network, Node, Region
from nupic.pynodes import *
from nupic.pynodes.ImageSensor import *
from nupic.pynodes.PyNode import *
from nupic.network import SimpleFanIn, SimpleSensorLink, CreateNode, SingleLink
from nupic.network.nodetools import \
  VectorFileSensorTool, VectorFileEffectorTool, \
  Zeta1NodeTool, Zeta1TopNodeTool, GetNTAPlugins

def createNetwork(untrainedNetwork):

  # Create empty network
  net = Network()

  # Set image width and height
  imageWidth = 64
  imageHeight = 32

  # Create sensor node
  sensorNode = CreateNode("nupic.pynodes.ImageSensor.ImageSensor",
    phase=0, image=(imageWidth*imageHeight) , category=1,
    imageWidth=imageWidth, imageHeight=imageHeight)

  # Add sensor to the network
  net.addElement("sensor",sensorNode)

  # Create first level template node
```

```
templateNodeLevel1 = Zeta1NodeTool(phase=1,
  poolerAlgorithm='gaussian',
  symmetricTime=False,
  transitionMemory=5,
  topNeighbors=1,
  maxGroupSize=1,
  grouperAlgorithm='sumProp',
  outputElementCount=200000,
  maxDistance=4624,
  detectBlanks=1)

# Create first level of nodes
myRegion = Region([16,32], templateNodeLevel1)

# Add first level to network
net.addElement("Level1", myRegion)

# Link sensor to first level
net.link("sensor", "Level1", SimpleSensorLink([imageHeight,imageWidth],"image",
"bottomUpIn"))

# Create second level top node
topnode = CreateNode("Zeta1TopNode",
  phase=2,
  categoriesOut=11,
  mapperAlgorithm='sumProp',)

# Add top node
net.addElement("Level2", topnode)

# Link first level to top level
```

```
  net.link("Level1", "Level2", SimpleFanIn())

  # Link sensor to top level (for sending the category of the training pattern)
  net.link("sensor", "Level2", SingleLink("category", "categoryIn"))

  # Create effector for output to a file
  net.addElement("FileOutput", VectorFileEffectorTool(phase=3))

  # Link category outputs and top node output to the file writing effector
  net.link("Level2", "categoriesOut", "FileOutput", "dataIn")
  net.link("sensor","category","FileOutput","dataIn")

  # save the network in a file
  net.writeXML(GetNTAPlugins(), untrainedNetwork)
  print "Network created."

#===============================================================================
# If invokedfrom the command line, just create network and save it
#===============================================================================
if __name__== '__main__':
    createNetwork('speech.xml')
    print "Saved HTM network to file 'speech.xml'"
```

A.6 TrainNetwork.py for the optimized HTM network

```
#!  /usr/local/bin/python

from nupic.session import Session
import sys
import os

def trainNetwork(untrainedNetwork, trainedNetwork):

  # Prepare session
  session = Session(os.path.join("sessions", "speech"))
  session.start()

  # Load network file and set parameters
  session.loadNetwork(untrainedNetwork)
  print 'Network loaded.'

  # Set sigma
  session.execute('Level1.*', ['setParameter', 'sigma', '159'])

  # Set type of sweeps the sensor must make (only to the right)
  session.execute("sensor", "self.sweepTypes=['right']")

  # Directory containing 11 directories.  Each of these 11 directories contains
  all train patterns of a certain digit
  trainDirectory =
  ""

  # Load BMP images
  session.evalPyNodeExpr("sensor", "self.loadMultipleImages('"+ trainDirectory
```

```
+
  "', 'bmp')")

  #--------------------------------------------------------
  # Train Level 1
  #--------------------------------------------------------

  print 'Training level 1...'

  # Set the width and height of the sensor window
  session.execute('sensor', 'self.enabledWidth = 2')
  session.execute('sensor', 'self.enabledHeight = 32')

  # Reset sensor
  session.evalPyNodeExpr('sensor', 'self.reset()')

  # Disable all nodes
  session.disableNodes()

  # Compute number of iterations
  iterations = session.evalPyNodeExpr('sensor','self.numIterations()')

  print 'Number of iterations:', iterations, '.'

  # Enable sensor
  session.enableNodes("sensor")

  # Enable one column of nodes
  session.enableNodes("Level1\[.*,0]")

  # Set column of nodes to learning mode
```

```
session.execute('Level1\[.*,0]', ['setLearning', '1'])

# Train the column of nodes on the train patterns
session.compute(iterations)

# Unset column of nodes from learning mode
session.execute('Level1\[.*,0]', ['setLearning', '0'])

# Clone every node state in the column to the nodes in its row
session.sendRequest('nodeCopyState Level1\[0,0] Level1\[0.*]');
session.sendRequest('nodeCopyState Level1\[1,0] Level1\[1.*]');
session.sendRequest('nodeCopyState Level1\[2,0] Level1\[2.*]');
session.sendRequest('nodeCopyState Level1\[3,0] Level1\[3.*]');
session.sendRequest('nodeCopyState Level1\[4,0] Level1\[4.*]');
session.sendRequest('nodeCopyState Level1\[5,0] Level1\[5.*]');
session.sendRequest('nodeCopyState Level1\[6,0] Level1\[6.*]');
session.sendRequest('nodeCopyState Level1\[7,0] Level1\[7.*]');
session.sendRequest('nodeCopyState Level1\[8,0] Level1\[8.*]');
session.sendRequest('nodeCopyState Level1\[9,0] Level1\[9.*]');
session.sendRequest('nodeCopyState Level1\[10,0] Level1\[10.*]');
session.sendRequest('nodeCopyState Level1\[11,0] Level1\[11.*]');
session.sendRequest('nodeCopyState Level1\[12,0] Level1\[12.*]');
session.sendRequest('nodeCopyState Level1\[13,0] Level1\[13.*]');
session.sendRequest('nodeCopyState Level1\[14,0] Level1\[14.*]');
session.sendRequest('nodeCopyState Level1\[15,0] Level1\[15.*]');

# Enable all nodes on the first level
session.enableNodes("Level1.*")

# Set column of nodes to inference mode
session.execute('Level1.*', ['setInference', '1'])
```

```
#--------------------------------------------------------
# Train Level 2
#--------------------------------------------------------

print 'Training level 2...'

# Set the width and height of the sensor window
session.execute('sensor', 'self.enabledWidth = 64')
session.execute('sensor', 'self.enabledHeight = 32')

# Reset sensor
session.evalPyNodeExpr('sensor', 'self.reset()')

# Compute number of iterations
iterations = session.evalPyNodeExpr('sensor','self.numIterations()')

print 'Number of iterations:', iterations, '.'

# Enable top node
session.enableNodes("Level2.*")

# Enable one column of nodes
session.execute('Level2.*', ['setLearning', '1'])

# Train the top node on the train patterns
session.compute(iterations)

# Unset top node from learning mode
session.execute('Level2.*', ['setLearning', '0'])
```

```
  # Set top node to inference mode
  session.execute('Level2.*', ['setInference', '1'])

  # Enable all nodes
  session.enableNodes()

  #---------------------------------------------------------------
  # Save the network and retrieve trained network file from bundle
  #---------------------------------------------------------------

  print 'Training complete.'

  # Save the trained network
  session.saveRemoteNetwork(trainedNetwork)

  # Stop session
  session.stop()

  # Retrieve the network file
  session.getLocalBundleFiles(trainedNetwork)

  # Clean up unused files
  session.setCleanupLocal(True)

#==============================================================================
# If invoked from the command line, do nothing
#==============================================================================
```

Bibliography

[1] J. Hawkins and S. Blakeslee, *On Intelligence*. Henry Holt, New York, 2004.

[2] D. George and J. Hawkins, "A hierarchical bayesian model of invariant pattern recognition in the visual cortex," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN-05)*, 2005.

[3] R. G. Leonard and G. Doddington, *TIDIGITS*. Linguistic Data Consortium, Philadelphia, 1993.

[4] R. K. Moore, "A comparison of the data requirements of automatic speech recognition systems and human listeners," in *Proceedings of the 8th European Conference on Speech Communication and Technology, Eurospeech 2003: Genève*, 2003, pp. 2582–2584.

[5] D. Hubel and T. Wiesel, "Functional architecture of macaque monkey visual cortex (ferrier lecture)," in *Proceedings of the Royal Society of London (Biology) 198*, 1977, pp. 1–59.

[6] V. Mountcastle, "An organizing principle for cerebral function: the unit model and the distributed system," in *The Mindful Brain*, G. Edelman and V. Mountcastle, Eds. MIT Press, 1978.

[7] R. D. Patterson *et al.*, "Complex sounds and auditory images," in *Acoustical Signal Processing in the Auditory System*, ser. Advances in the Biosciences. Pergamon Press, 1992, vol. 83, pp. 429–433.

[8] E. de Boer and H. de Jongh, "On cochlear encoding: potentialities and limitations of the reverse-correlation technique," *Journal of the Acoustical Society of America*, vol. 63, pp. 115–135, 1978.

[9] L. Carney and C. Yin, "Temporal coding of resonances by low-frequency auditory nerve fibers: Single fibre responses and a population model," *Journal of Neurophysiology*, vol. 60, pp. 1653–1677, 1988.

[10] B. Moore, R. Peters, and B. Glasberg, "Auditory filter shapes at low center frequencies," *Journal of the Acoustical Society of America*, vol. 88, pp. 132–140, 1990.

[11] M. Slaney, "Auditory toolbox: A MATLAB toolbox for sound," Apple Technical Report #45, Tech. Rep., 1994.

[12] R. Meddis, "Simulation of mechanical to neural transduction in the auditory receptor," *Journal of the Acoustical Society of America*, vol. 79, pp. 702–711, 1986.